UNIVERSITY OF CALIFORNIA
Santa Barbara

Memory Page Stability and its Application to Memory Deduplication

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Karim Yehia Elghamrawy

Committee in Charge:

Professor Frederic T. Chong, Chair

Professor Diana Franklin

Professor Timothy Sherwood

June 2016

The Dissertation of
Karim Yehia Elghamrawy is approved:

Professor Diana Franklin

Professor Timothy Sherwood

Professor Frederic T. Chong, Committee Chairperson

May 2016

Memory Page Stability and its Application to Memory Deduplication

by

Karim Yehia Elghamrawy

Dedicated to my parents, Wafaa and Yehia!

Acknowledgements

First and foremost, I would like to thank all my committee members whose help, support, scholarly insights, and intellectual contributions were crucial to the completion of this dissertation. My supervisor, *Professor Chong*, helped me immensely during my Ph.D journey. He always identified potential problems and suggested solutions from a high-level perspective. This perspective is very important for a Ph.D student because it can save months or even years of toiling away in the wrong direction. This high-level perspective was complemented by *Professor Franklin*'s attention to low-level details. This low-level perspective is also important because this is the level where the feasibility and efficiency of a good high-level idea is eventually determined. *Professor Sherwood*'s excellent mentorship, helpful tips, and thorough feedback during my major area examination(MAE) and my proposal are only surpassed by his friendly and cheerful character.

I would like to thank my *parents* who taught me from an early age to work hard, be persistent, and not to break down in the face of pressure. All of these traits are of extreme importance in the lifetime of a Ph.D student. Thank you *Wafaa Raslan* and *Yehia Elghamrawy* for your unconditional love and infinite support. I would also like to thank my sisters, *Haidy*, *Norhan*, and *Miral* for their love, support, and care.

I am also very grateful to the students of ArchLab. We exchanged ideas, discussed problems, analyzed solutions, and helped each other along the way. Thank you *Zhaoxia Deng*, *Ying Gao*, and *Weilong Cui* for the fruitful discussions. Special thanks to *Lunkai Zhang* for his contribution to this dissertation through his expertise with architectural simulators.

Emotional support is pivotal for a Ph.D student. There are many stressful moments that face Ph.D students: approaching deadlines, problems that appear unexpectedly, etc. All these sporadically-occurring problems can take its toll on a Ph.D student. Emotional support from close people help tremendously. I would like to thank *Emilia Hannuksela* for her limitless support, capacity to listen, and her empathetic character that gave me the unflagging strength to persevere.

Support from friends has come in different forms and shapes. I would like to specifically thank *Andrew Irish*, *Stratos Dimopoulos*, *Alexander Pucher*, *Theodore Georgiou*, *Alex Kermani*, *Kyle Clark*, *William Miller*, *Victor Zachary*, *Mai Elsherif*, *Hassan Wassel*, *Stefan Rasche*, and *Francesco Di Massa*.

This work would not have been possible without all of you. Thank you very much!

Curriculum Vitæ

Karim Yehia Elghamrawy

Research Interests

I am interested in Virtualization, Cloud Computing, and Distributed Systems.

Education

2016	Ph.D. in Computer Science, University of California, Santa Barbara.
2015	Masters of Science in Computer Science, University of California, Santa Barbara.
2011	Masters of Science in Electrical and Computer Engineering, Cairo University.
2008	Bachelors of Science in Electrical and Computer Engineering, Cairo University.

Professional Experience

2015	Software Engineering Intern at VMware
2014	Software Engineering Intern at Appfolio
2011 - 2016	Research Assistant at the CS department, UCSB
2011 - 2012	Teaching Assistant at the CS department, UCSB
2008 - 2011	Teaching Assistant at the ECE department, Cairo University

Abstract

Memory Page Stability and its Application to Memory Deduplication

Karim Yehia Elghamrawy

In virtualized environments, typically cloud computing environments, multiple virtual machines run on the same physical host. These virtual machines usually run the same operating systems and applications. This results in a lot of duplicate data blocks in memory. Memory deduplication is a memory optimization technique that attempts to remove this redundancy by storing one copy of these duplicate blocks in the machine memory which in turn results in a better utilization of the available memory capacity.

In this dissertation, we characterize the nature of memory pages that contribute to memory deduplication techniques. We show how such characterization can give useful insights towards better design and implementation of software and hardware-assisted memory deduplication systems. In addition, we also quantify the performance impact of different memory deduplication techniques and show that even though memory deduplication allows for a better cache hierarchy performance, there is a performance overhead associated with copy-on-write exceptions that is associated with diverging pages.

We propose a generic prediction framework that is capable of predicting the stability of memory pages based on the page flags available through the Linux kernel. We evaluate the proposed prediction framework and then discuss various applications that can benefit from it, specifically memory deduplication and live migration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we give a brief introduction to virtualization and memory deduplication. We also present the outline of the dissertation and our contributions.

1.1 Hardware Virtualization

Hardware virtualization is the technology that allows the creation of virtual machines. These virtual machines act as if they are real physical machines. They run on top of a software layer called the virtual machine monitor or the hypervisor.

This software layer separates the virtual machines from the underlying hardware resources which allows for better utilization of the hardware resources, better isolation among virtual machines, easier and faster portability, and fault tolerance. All of these features made virtualization the enabling technology of *cloud computing*.

There are different types of hardware virtualization:

- *Full virtualization*: The guest operating system is not aware that it is running in a virtualized environment and it runs unmodified on the virtual machine.

- *Paravirtualization*: The guest operating system is aware that it is running in a virtualized environment. It is modified in a way that allows better performance in such environments.

1.2 Memory Deduplication

In virtualized environments, typically cloud computing environments, there are multiple virtual machines that run on the same physical host. These virtual machines often run similar operating systems and applications which may result in a lot of identical data blocks in the machine memory.

Efficient management of the available memory system is crucial. The memory system dominates the cost of the whole system and it consumes the majority of the power. Memory deduplication allows for efficient utilization of the available memory capacity by storing only one copy of the duplicate memory blocks in the machine memory.

Due to its effectiveness, memory deduplication techniques are widely used in popular hypervisors, and heavily studied in research. In the next chapter, we discuss in detail how memory deduplication works.

Specifically, memory deduplication allows for:

- Better utilization of the virtual machine memory. For example, all zero pages are reduced to only one zero page in the host memory allowing other virtual machines to use the available memory that is not otherwise used.
- Better consolidation since reclaiming memory will allow more virtual machines to run on the same host. This increases the processor utilization and the overall performance of the system.

1.3 Thesis Statement and Dissertation Roadmap

Data similarity in virtualized environments can be exploited to use the available memory capacity efficiently if the data is relatively stable. We investigate the nature of this similarity and propose a generic prediction framework that can efficiently predict relatively stable pages.

This dissertation presents the following contributions:

1. We characterize the nature of identical and similar memory pages based on the origin and stability of these pages. We explore the importance of

page stability for memory deduplication and we also characterize and quantify the different performance aspects associated with memory deduplication techniques.

2. After discussing the importance of page stability for memory deduplication, we propose a generic prediction framework that can predict the stability of memory pages based on the page flags that are already available through the Linux kernel.

3. We explore, study, and evaluate some applications that can benefit from the proposed prediction framework. Specifically, we thoroughly investigate how the proposed prediction framework can improve memory deduplication and live migration techniques.

The rest of the dissertation is laid out as follows: Chapter 2 gives an overview of various memory deduplication techniques and characterizes the nature of data similarity in memory and the performance impact of memory deduplication. In chapter 3, we characterize the stability of memory pages based on their kernel page flags and propose a generic stability prediction framework that can predict relatively stable memory pages. In chapter 4, we discuss and evaluate some applications that can benefit from the prediction framework proposed. Finally, Chapter 5 concludes our work and discusses future work.

Chapter 2

Characterization of Memory Similarity in Virtualized Environments

"*Share our similarities, celebrate our differences.*"

— *M. Scott Peck*

Memory systems dominate the system cost so it is crucial to utilize the available memory capacity efficiently. In virtualized environments, virtual machines running on the same physical host often run identical or similar operating systems and applications. This results in identical blocks of data in the physical memory. Memory deduplication is a widely used technique in virtualized environments that is used to reduce the memory footprint of virtual machines by eliminating these redundant blocks and storing only one copy of them in the machine memory. In this chapter, we give a general overview of software-based and hardware-assisted

memory deduplication techniques, characterize the nature of the pages involved in memory deduplication, and quantify the performance impact of such techniques.

2.1 Introduction

The memory system is generally the most expensive component of computing systems and hence an efficient utilization of the available memory capacity is very important. Memory deduplication is a memory optimization technique that tries to reclaim memory capacity by sharing identical blocks of data. Memory deduplication is already adopted by popular hypervisors [4, 103] and there is a lot of work proposed that aims at extending its functionality or enhancing its performance [70, 69, 20, 33, 91, 71]. Some of the ideas suggested are based on software only [70, 69, 20, 33, 71]. Others rely on hardware assistance [91]. We believe that a deeper understanding and characterization of the nature of these identical memory blocks can provide some useful insights that can be exploited to enhance the design and implementation of memory deduplication techniques.

The rest of the chapter is laid out as follows: Section 2.2 gives a brief overview of memory deduplication. In section 2.3, we characterize the nature of identical memory blocks in virtualized environments. Section 2.4 characterizes the performance implications of memory deduplication. Finally, section 2.5 concludes this chapter.

2.2 Background

This section gives an overview of memory deduplication techniques. Memory deduplication techniques can be classified based on their implementation to *software-only* and *hardware-assisted* techniques. They can also be classified based on the granularity of sharing. The most popular and commercially adopted deduplication techniques use page-size granularity for sharing due to the simplicity of its implementation. However, sub-page granularity might lead to more memory savings than a full-page granularity at the expense of a more complex implementation. In this section, we will introduce a candidate memory deduplication technique for each of the aforementioned classifications: a software-based memory deduplication at a page granularity, a software-based memory deduplication at a sub-page granularity, and a hardware-assisted memory deduplication. We start by discussing *content-based page sharing* which is the most widely used technique for memory deduplication and it relies on sharing identical pages. Afterwards, we talk about how the idea of sharing identical pages was extended in the *Difference Engine* [33] to allow for sharing pages at a sub-page granularity. After we discuss these software-based mechanisms and illustrate their advantages and shortcomings, we talk about *Page Overlays* [91] which is a hardware-assisted mechanism that was proposed to alleviate the shortcomings of software-only memory deduplication.

2.2.1 Content-based memory sharing

Content-based memory sharing is a memory deduplication technique that was introduced by VMware [103]. The basic idea is to identify duplicate memory pages at run time through the hash values of the content of the page [35, 36]. The hash value is used to index a hash table that is maintained by the VMware ESX server. If a collision happens, this indicates that there is a possibility that the pages in the hash table entry can be identical to the colliding page. A byte-by-byte comparison ensues to detect duplicate pages. Figure 2.1 illustrates the idea of content-based memory sharing.

If a duplicate page is found, then a page can be shared by updating the page table such that all identical pages are mapped to the same machine address. Moreover, these pages are also marked as read-only. Any writes to shared pages are handled by a copy-on-write exception.

The basic idea of copy-on-write was introduced in [11]. The general idea of copy-on-write is to share identical objects instead of creating identical copies. One of the most popular applications of the copy-on-write technique in the Linux operating system [32, 62, 14, 98] is the *fork* system call. The *fork* system call is used to create a process that is an exact copy of the process that is executing the fork. The created process is called the *child* process and the process creating it is called the *parent* process. To speed up the creation of the child process, Linux

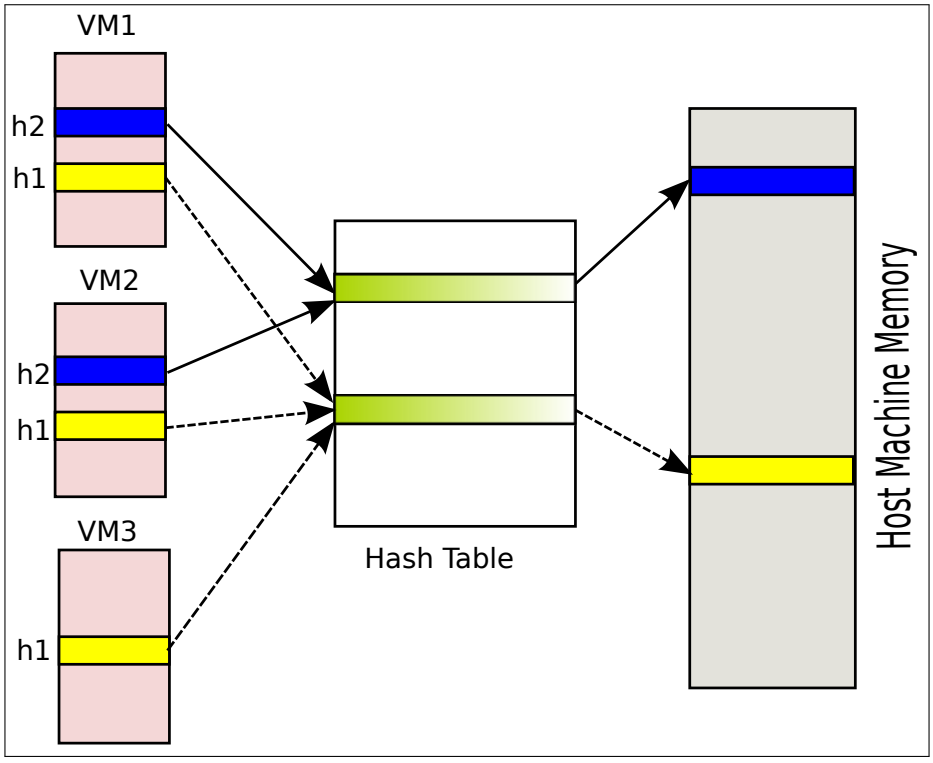


Figure 2.1: Memory Deduplication using a hash table. The content of the page is hashed and checked against a hash table to identify duplicate pages. In the figure, three pages have a hash h1 and two pages have a hash h2. If a duplicate is found, the hypervisor maps the duplicate pages into the same host machine memory frame. The figures show five pages from three different virtual machines reduced to only two pages in the machine memory.

shares the memory pages of the parent with the child by mapping the virtual pages of the child to the same physical pages of the parent. These shared pages are further marked as read-only. Any updates to these pages is handled by a copy-on-write exception where a new private page is allocated and a corresponding page table update is performed.

In [103], it was shown that sharing identical pages can reclaim up to 40% of the machine memory for homogeneous workloads. Content-based memory sharing is also used [4, 52, 20] in other open source hypervisors like Xen [6] and KVM [50].

One of the shortcomings of the copy-on-write mechanism is the performance penalty of copy-on-write exceptions. A copy-on-write exception is handled in two steps. First, a new free physical page is identified and then the content of the original page is copied to the new page. Second, the virtual memory page that received the write is remapped to the newly allocated physical page. Both steps incur high latency and are on the critical path [10, 90, 87, 99, 101]. The copy operation consumes high memory bandwidth [90] and remapping typically requires a TLB shootdown [10, 99].

2.2.2 Difference Engine

We introduce the *Difference Engine* [33] as a candidate software-based memory deduplication technique that aims at sharing memory at a sub-page granularity.

The *Difference Engine* tries to extend content-based memory sharing by additionally sharing pages that are *similar*. Sharing similar pages is performed by patching these pages against a reference page. A page is said to be *similar* to another page if the patch size is less than half the size of the page. In addition to patching similar pages, pages that have not been accessed for a long time are further compressed. Figure 2.2 shows how the *Difference Engine* reclaims memory through identical page sharing, similar page sharing, and compression.

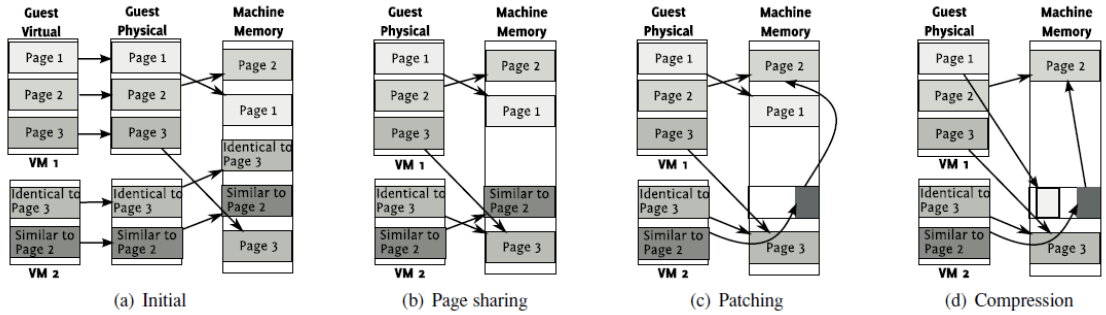


Figure 2.2: The memory reclamation techniques employed by Difference Engine: identical page sharing, similar page sharing through page patching, and compression. In this example, five physical pages are stored in less than three machine memory pages for a savings of roughly 50% [33]

Identical page sharing is performed exactly as described previously. Memory pages are scanned and hashed. Based on the hash value, an entry in the hash table is checked for any collisions. If a collision occurs, a byte-by-byte comparison is performed to detect if the page is identical. In the case of identical page detection,

the page is merged and the page table entry for this page is updated such that it points to the same machine memory address. The page is also marked as read only and writes are handled through copy-on-write exceptions.

Similar page sharing is performed by maintaining another hash table that is exclusively used for identifying similar pages. The *Difference Engine* hashes two 64-byte blocks at two fixed locations in the page. Candidate similar pages are detected by hashing the 64-byte blocks at these fixed locations and detecting collisions.

Further, *memory compression* is used to compress pages that were not accessed for a very long time to further decrease the memory footprint.

As mentioned earlier, sharing identical and similar pages leads to reclaiming memory capacity at the expense of potential performance degradation due to the exceptions that occur when a merged identical page is written to, or a similar/compressed page is accessed (read or write). For these reasons, the *Difference Engine* uses a non-recently-used policy to choose pages that are good candidates for sharing. This is implemented by using the *Modified* and *Referenced* bits in the page table to track pages. All pages that are recently modified are ignored. Pages that are recently accessed but not modified are good candidates for sharing and being a reference page for similar pages. Pages that are not recently accessed are considered for patching or compression.

In summary, the *Difference Engine* allows for the possibility of achieving more memory savings by sharing memory at a finer granularity than full page sharing. However, the shortcomings of the idea are:

1. In addition to the copy-on-write exceptions associated with diverging identical pages, for patched pages even reads will cause an exception where the page has to be unpatched before the reading proceeds. This is not the case with identical page sharing where exceptions only happen in the occasion of write accesses.

2. Frequently modified pages are ignored. A frequently modified page that can benefit from sub-page granularity sharing (e.g. a page that frequently modifies a fixed byte within the page) will not be considered to be a candidate for sharing. This is mainly related to the restrictions enforced by the implementation.

To fix these shortcomings, hardware-assisted deduplication was proposed [91]. The main idea is to introduce some hardware modifications to manage memory at a finer granularity. With hardware support, the performance overhead of copy-on-write exceptions can be alleviated. This allows for sharing identical or similar pages that are frequently modified. In the next section, we give a quick overview of the *page overlays*.

2.2.3 Page Overlays

In general, managing virtual memory [24, 29, 49] at a finer granularity (e.g. cache granularity) can be useful for a multitude of applications. For example, these benefits include: 1) Eliminating or reducing the performance penalties associated with copy-on-write exceptions. 2) Fine-grained deduplication [21, 33] 3) Fine-grained data protection [106] 4) Compression at the cache block level [26, 43, 82].

The *page overlays* [91] provides a hardware framework that enables managing memory at a finer granularity. As shown in Figure 2.3, the framework allows a virtual page to be mapped to a physical page and an overlay. An overlay has a smaller size than a physical page and it contains only a subset of the cache blocks of the page. As Figure 2.4 illustrates, Cache blocks that exist in the overlay have higher precedence to the cache blocks existing in the physical page. That is, if a cache block is required to be accessed, the overlay is searched first. Only cache blocks that do not exist in the overlay are accessed through the regular physical page.

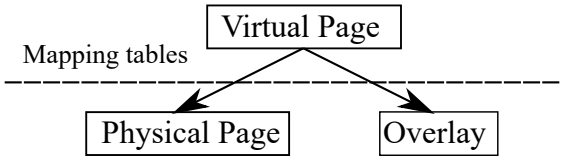


Figure 2.3: The basic idea of the *page overlays*. A virtual page can be mapped to a regular physical page and an overlay.

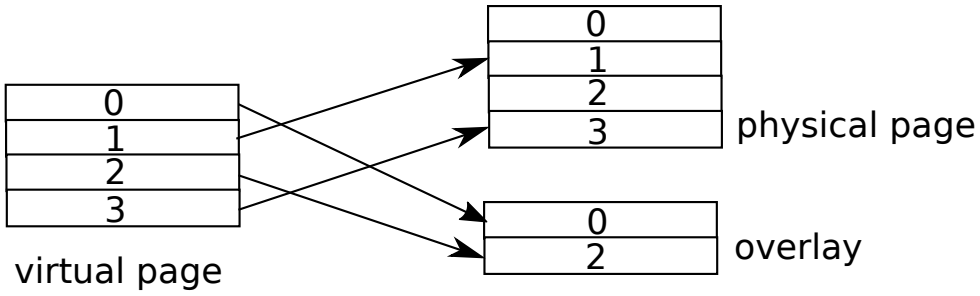


Figure 2.4: The access semantics of the page overlays. The figure shows an example of a page that consists of four cache blocks. Cache blocks that exist in the overlay are accessed first. Cache blocks in the physical page are only accessed when they do not exist in the overlay.

To check if a cache block is part of an overlay, the framework maintains a bit vector of each virtual page called the *overlay bit vector*. Each bit in the bit vector corresponds to a cache block and denotes whether the cache block is in the overlay or not. This bit vector is also cached in the TLB to allow for a fast detection.

If a cache block was found to be in the overlay of a page, the machine address of the cache block can be fetched through a dual-address mechanism. One address for the processor caches called the *Overlay Address* and another for the actual machine address which is called the *Overlay Memory Store Address*.

In the overlay address space, the size of an overlay is the same as the size of a virtual page. This address space is taken from the unused physical address space.

In this framework, the main memory stores both regular physical pages and overlays. The overlays are stored in an *overlay memory store* (OMS). The OMS stores the overlays in a compact way. Translating overlay addresses to real machine addresses in the OMS is done through a mapping table called the *Overlay Mapping Table* (OMT) which is stored in the memory controller. Figure 2.5 shows how virtual memory is managed in the page overlays framework.

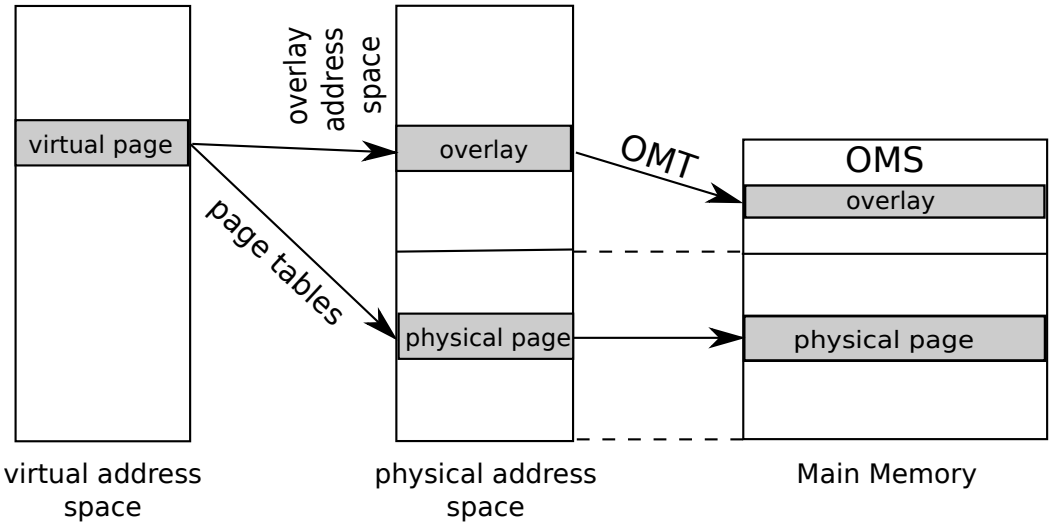


Figure 2.5: The Figure shows how virtual memory is managed in the page overlays framework. The overlays address space is part of the unused physical address space. Overlays are mapped to the OMS through a mapping table, OMT, which is stored in the memory controller.

In [91], it was suggested to use the page overlays framework for fine-grained memory deduplication. This can be achieved by storing the differing cache blocks

in the overlays. This technique has two advantages over the software implementations [33]. First, instead of patching, similar pages can just store their different cache blocks in the overlays memory store. This allows for seamless penalty-free access to the differing cache blocks whether the access is a read or write without the performance hit caused by unpatching or copy-on-write exceptions. Second, unlike software implementations, pages that are frequently modified can contribute to further memory savings if the modification is limited within a few cache blocks.

All of the above requires a deeper understanding and a thorough characterization of the nature of identical and similar pages. This deep understanding might give more insights about the usefulness of using these systems and/or better design or improved implementations of these systems. In the following sections, we will try to analyze, characterize, and understand the nature of memory pages that contribute to memory savings through deduplication.

2.3 Page Characterization

In this section, we try to characterize the nature of *identical* and *similar* memory pages. We assume a page size of 4 KB. We divide each page into 64 64-byte blocks which is a typical cache granularity. *Identical* pages are pages that share the same content. That is, all blocks of identical pages have the same data. We

Table 2.1: Virtual Machine Configuration

VM1	Ubuntu 14 running the apache benchmark
VM2	Ubuntu 13 running the apache benchmark
VM3	Ubuntu 14 compiling the Linux kernel
VM4	Ubuntu 14 running redis benchmark
VM5	Ubuntu 14 running sysbench

define *similar* pages to be pages that have more than 32 blocks that have the same data. In other words, similar pages have less than 32 differing blocks.

We start by introducing our proposed workload mixes that we use to characterize the desired pages. Table 2.1 shows all the virtual machines, the operating systems running on them, and the actual benchmark that is running on top of the operating system. Each virtual machine is configured with 1 vCPU and 512 MB of memory. Each workload mix is a combination of two virtual machines running on the same host on top of KVM. Table 2.2 shows all the workload mixes. Workload mixes 1, 2, and 3 are the same as mixes 4, 5, and 6 with the only exception of VM2 running on Ubuntu 13 instead of Ubuntu 14. The reason is we want to test how introducing a slight operating system heterogeneity (a version change of Ubuntu) will impact our characterization results.

Table 2.2: Workload Mixes

Mix1	VM1 and VM3
Mix2	VM1 and VM4
Mix3	VM1 and VM5
Mix4	VM2 and VM3
Mix5	VM2 and VM4
Mix6	VM2 and VM5
Mix7	VM3 and VM5
Mix8	VM4 and VM5

2.3.1 Percentage of Identical and Similar Pages

We start by showing the percentage of identical and similar memory pages for each of the workloads of Table 2.2. Figure 2.6 shows, for each workload, the percentage of memory pages that are identical and those that are similar to at least one other memory page. Even though Mixes 1,2, and 3 run the same applications as Mixes 4, 5, and 6, it is obvious that introducing a small OS version heterogeneity to the system decreases both the number of identical pages and similar pages, which in turn is expected to reduce memory savings. Mix 7 has the largest percentage of identical pages while Mix 8 has the largest percentage of similar pages.

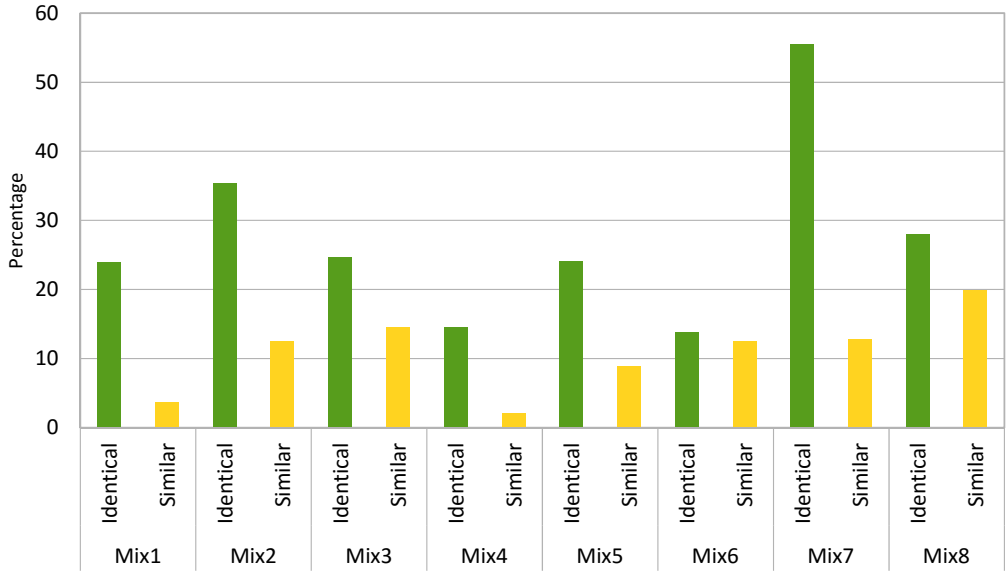


Figure 2.6: This figure shows the percentage of identical and similar pages in each of the workload mixes.

2.3.2 Origin-based Page Group Classification

To further study the origin of identical and similar pages, we define a *page group* to be a group of pages that are shared either because they share identical or similar content.

If all the pages contained within a page group originate from the same virtual machine, we call this a *pure* page group. On the other hand, *hybrid* page groups

are groups that contain pages originating from different virtual machines. Barker et al. [7] did this study for identical page sharing and they concluded that the majority of page groups are generally *pure* unless the same OS is running on the virtual machines.

All the workloads were executed and a snapshot was taken shortly after the execution. We examined this snapshot to determine all the page groups and classify these page groups based on their origin to *pure* and *hybrid* page groups.

Figure 2.7 shows the classification of page groups containing *identical* pages. The results are consistent with the results observed by [7]. Even a slight operating system version heterogeneity makes almost all of the page groups *pure*. Hybrid page groups are significant only when the same version of the same operating system runs on the virtual machines.

We extend the study by investigating the purity of page groups containing similar pages. Figure 2.8 shows the pure and hybrid page groups that contain similar pages for each of the workload mixes. Page groups of similar pages behave similarly to page groups of identical pages. The majority of page groups are hybrid if the virtual machines are running the same version of the same operating system. The slightest version variation leads to making almost all page groups pure.

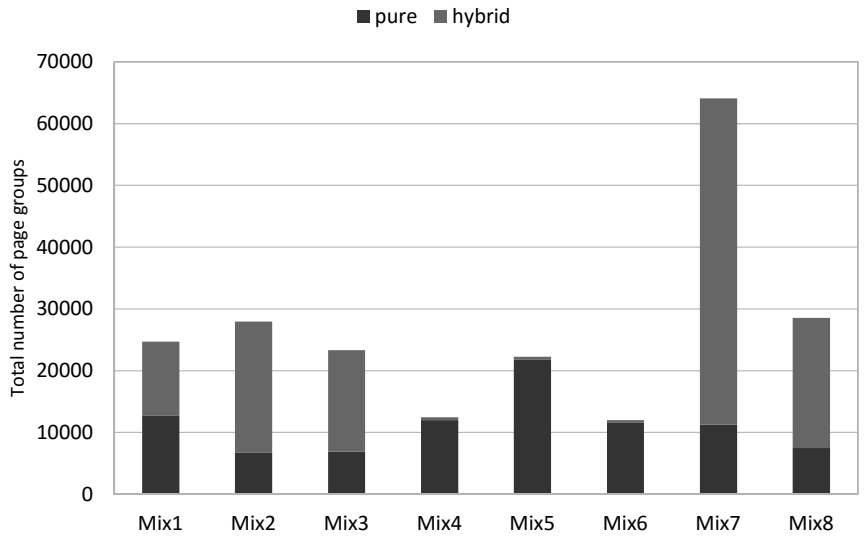


Figure 2.7: The total number of pure and hybrid page groups that contain *identical* pages for each workload mix.

2.3.3 Stability-based Page Classification

Another classification of identical and similar pages is based on their stability. By stability, we mean the property that a memory page will remain stable, or unchanged, for a period of time that is long enough for a full memory scan. In our work, we will assume this time period to be 5 minutes which we believe to be a reasonable time for a full scan of a memory of size 1 GB (about 3.5 MB/sec).

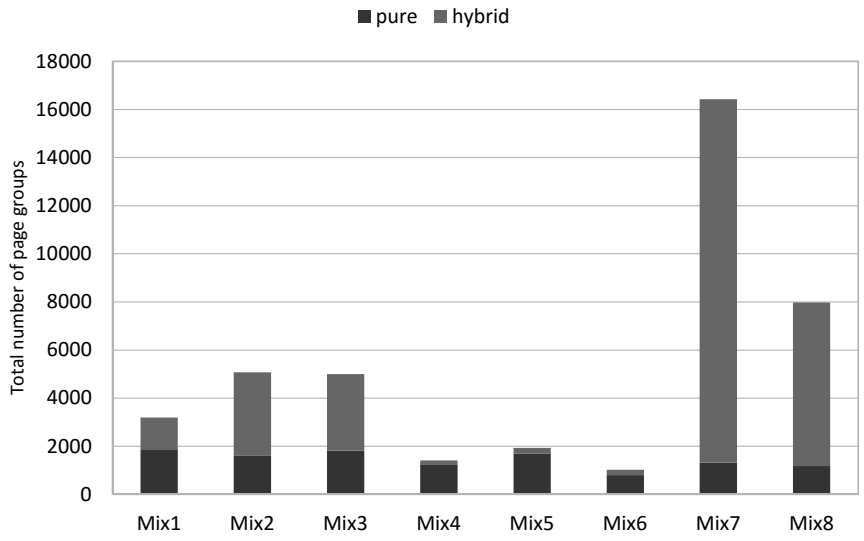


Figure 2.8: The total number of pure and hybrid page groups of *similar* pages for each workload mix.

As mentioned earlier, we divide a 4 KB page into 64 64-byte blocks. We classify pages based on their stability during a specified period of time into three different stability classes:

- *Stable pages* are the pages that remain completely unchanged.
- *Pseudo-stable pages* are pages that are not stable, but they change only slightly by less than 32 64-byte blocks.
- *Unstable pages* are pages that change by more than 32 64-byte blocks.

Stability-based classification of pages that contribute to memory deduplication is very useful at giving insights into the pages that are *actually* good candidates for sharing. For example, sharing a page for a very short time does not contribute to any real memory savings. Moreover, a diverging page will further result in an exception which is a performance overhead.

Stable pages are of paramount importance for software-based memory deduplication to achieve long-lasting memory savings and to avoid the negative performance impact of copy-on-write exceptions associated with writes to identical pages and writes/reads to similar pages. This is the only class of pages that is suitable for software-based memory deduplication. In hardware-assisted memory deduplication, stable pages are also good candidates for sharing but they are not the only good candidates.

Pseudo-stable pages are not good candidates for software-based memory deduplication due to the performance overhead discussed earlier. However, they are good candidates for hardware-assisted memory deduplication [91] especially when sharing identical pages.

Unstable pages are not useful for any sort of memory deduplication. They hurt both software and hardware implementations.

Table 2.3 summarizes how the stability of a page affects how good a page is for memory deduplication.

Table 2.3: Determining good page-sharing candidates based on the page stability class

class	*software deduplication*	*hardware-assisted deduplication*
Stable	Good	Good
Pseudo-stable	Bad	Good for identical pages. OK for similar pages
Unstable	Bad	Bad

In Figure 2.9, we try to investigate the stability of all identified identical and similar pages for each of our workloads. In this experiment, we first identify all identical and similar pages, then run the workloads for 5 minutes, and then classify the identified pages based on their stability.

We observe that for all workloads, the majority of identical pages tend to be *stable*. There is only a small number of *pseudo-stable* identical pages. Almost all identical pages that lose their stability become *unstable*. For similar pages, the majority of these pages end up either *stable* or *pseudo-stable*.

For hardware-assisted memory deduplication, the above observations indicate that pages that start out as identical will possibly not benefit from storing the differing blocks in overlays since these pages will likely end up being unstable.

For identical pages, it is more effective to avoid sharing pages that will eventually diverge rather than storing the differing blocks. On the other hand, the stability and pseudo-stability of similar pages make them very good candidates for hardware-assisted deduplication.

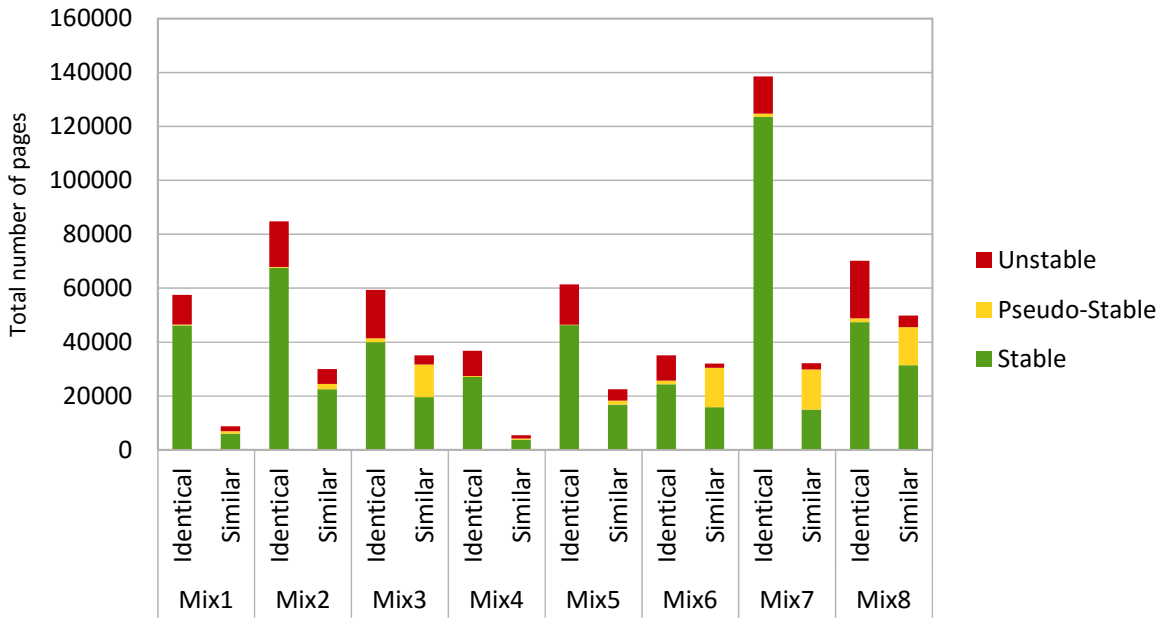


Figure 2.9: The total number of stable, pseudo-stable, and unstable identical and similar pages for each workload mix.

2.3.4 Characterizing Similar Pages

Here we try to focus more on studying similar pages. Shared similar pages will all have one reference page that is stored in the actual machine memory. The different cache blocks will be typically stored in a different storage. As we described earlier, in [91], the different cache blocks are stored in the main memory as well in the overlay memory store.

Similar pages contribute to memory savings when the number of differing cache blocks is relatively small. Therefore an important feature of a similar page is the number of cache blocks that are different from the reference page.

In Figure 2.10, we show the total number of similar pages for all workloads and classify them based on the number of cache blocks (d) that are different from their corresponding reference pages. We call d the *divergence size*. Similar pages with small divergence sizes contribute to memory savings more than similar pages with high divergence sizes.

2.3.5 Block Threshold Analysis

We define *block threshold* to be the maximum number of allowable differing blocks per page. For an infinite capacity of the storage where these differing blocks can be stored, a higher block threshold is always better because a higher block

threshold will lead to more memory saving. So as the block threshold increases, the memory savings increases.

However, in reality the storage where these differing blocks are stored is finite. Having higher threshold in this case will not necessarily mean more savings. For example, filling this storage with pages of low divergence size is more useful than filling it with pages of high divergence size.

For a given specific block threshold, We define efficiency η to be the ratio of the absolute memory savings achieved for this block threshold to the storage size required for the savings. A higher η means that the storage has, on average, a lower divergence size per page which is good for memory savings. We also define *savings* (S) to be the absolute memory saving due to similarity for this particular block threshold. As block threshold increases, (S) is expected to increase, but (η) is expected to decrease. Since both memory savings and the storage size are important factors, we plot the product of the two metrics η and S. Figure 2.11 shows the efficiency-savings ($\eta - S$) product for each mix of workloads. All workloads show a peak around approximately 4 blocks. To put things in perspective, Figures 2.12 and 2.13 show, for each workload, the memory savings achieved at different block thresholds and the required storage size required to store the differing blocks respectively. For the majority of workloads, a block threshold of 4 or 8 leads to reasonable memory savings with a small storage size. Increasing the

block threshold to 32 results in a very big increase in the storage size requirements without a proportional increase in savings. For example, in Mix 6, the storage size requirements for a block threshold of 32 is up to 35 times the size requirements of 8 blocks with only a 3.5x increase in memory savings.

2.4 Performance Characterization

In this section, we try to characterize the performance impact of sharing identical and similar pages. As discussed earlier, sharing pages has a performance overhead associated with the exceptions that happen in the occurrence of a write to an identical page or a read/write access to a patched (similar) page in the software-based memory deduplication.

We use simics [64] which is a full system simulator to evaluate the performance characterization of three different scenarios.

1. baseline: no sharing of identical or similar pages.

2. traditional: sharing identical pages only.

3. diffEngine: sharing identical and similar pages like the *Difference Engine*.

We run the Xen hypervisor [6] on top of simics and then we run three virtual machines on top of Xen. Each VM is configured with 1 GB of memory. Simics

is configured with 4 GB of memory. The simics source code was modified to implement the above scenarios. All the three VMs running on top of Xen run the same benchmark. We have three workload mixes.

- Sysbench: the sysbench benchmark
- GemsFDTD: a SPEC CPU2006 benchmark
- CactusADM: a SPEC CPU2006 benchmark

We simulate each of the above workloads for 24 seconds on simics and report our results. Figure 2.14, Figure 2.15, and Figure 2.16 show the memory distribution of these workloads between private pages, shared pages, and patched pages (for the diffEngine implementation).

Figure 2.17 shows the miss rate of the last-level cache for each of the workloads for the baseline, traditional, and the diffEngine scenarios. Even though sharing pages lead to a performance overhead in the occasion of a diverging page, it has a good impact on the cache hierarchy performance. That is because all of the shared pages will share the same space in the cache hierarchy leaving more space for other cache blocks. This results in a decrease in the miss rate. As the figure shows, the CactusADM shows a significant improvement in the LLC cache performance for the traditional and diffEngine scenarios relative to the baseline case. This

is because CactusADM has the highest percentage of *shared* pages as shown in Figure 2.16.

To characterize the performance impact of copy-on-write exceptions, we introduce the *ideal* scenario. In the ideal scenario, identical and similar pages are shared like the diffEngine scenario. However, the side effects of copy-on-write exceptions are ignored. Figure 2.18 shows the normalized IPC for each of the workloads for all the mentioned scenarios. The difference of performance between traditional/diffEngine and ideal denotes the performance lost due to copy-on-write exceptions. The results show that the performance overhead can be significant. For example, copy-on-write exceptions result in a performance reduction by 6.3% and 7.1% for GemsFDTD and CactusADM respectively.

We also observe that for cactusADM, the IPC of the traditional and diffEngine scenarios are significantly better than that of the baseline. This is due to the enhanced LLC performance of CactusADM.

2.5 Conclusion

In this chapter we provide an overview of all related work to memory deduplication. We explain how characterizing memory pages can provide some insights towards better design and/or implementation of memory deduplication systems.

We also characterize the performance issues related to memory deduplication. Based on our previous study, we can draw the following conclusions:

- A slight heterogeneity in the operating system will lead to a significant decrease in the number of identical and similar pages identified. Placing homogeneous VMs on the same physical host is advantageous for memory savings.

- In heterogeneous VMs, almost all of the page groups are pure.

- For hardware-assisted deduplication, having a block threshold for the number of the differing blocks is important and our analysis shows that the best block threshold is around 4 blocks for all the studied workloads.

- The majority of pages that start out as identical remain stable.

- The majority of diverging identical pages become completely unstable. This means these pages will not benefit much from hardware-assisted deduplication.

- The majority of similar pages are stable or pseudo-stable. This means similar pages can benefit from hardware assisted deduplication, especially if these pages are frequently read from.

- Software-based memory deduplication techniques suffer from the performance overhead of copy-on-write exceptions which can be significant.

- Memory deduplication allow multiple cache blocks to share the same cache space giving more free space for other cache blocks. This leads to an increase in the performance of the cache hierarchy.

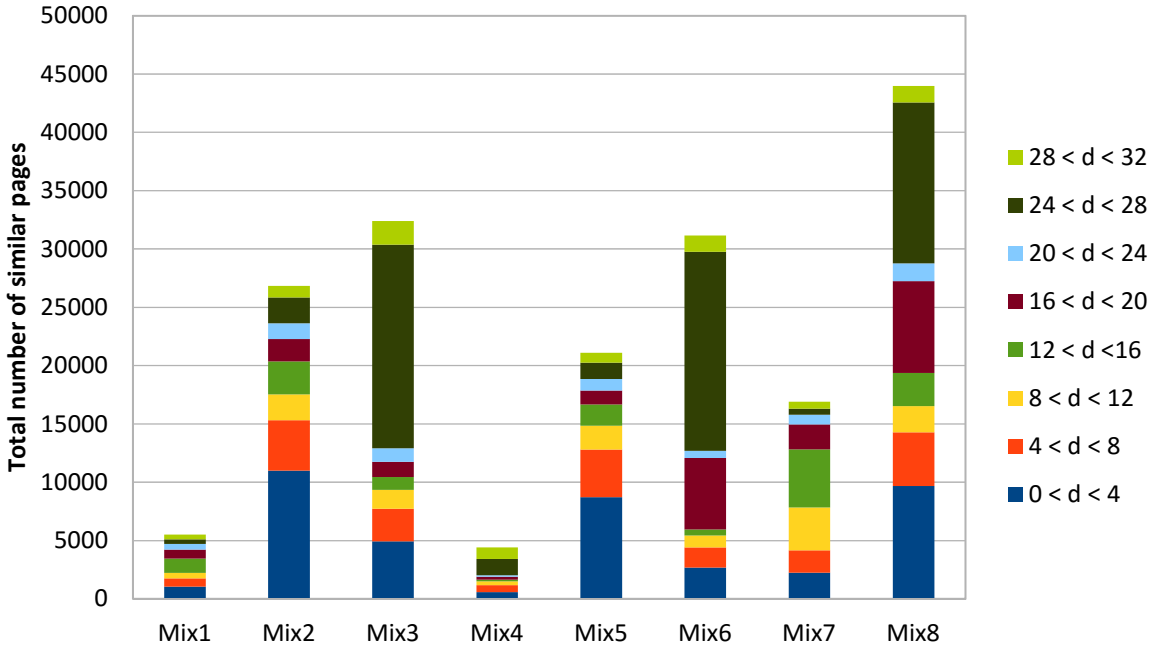


Figure 2.10: The distribution of similar pages based on the number of differing blocks (d) relative to their reference page.

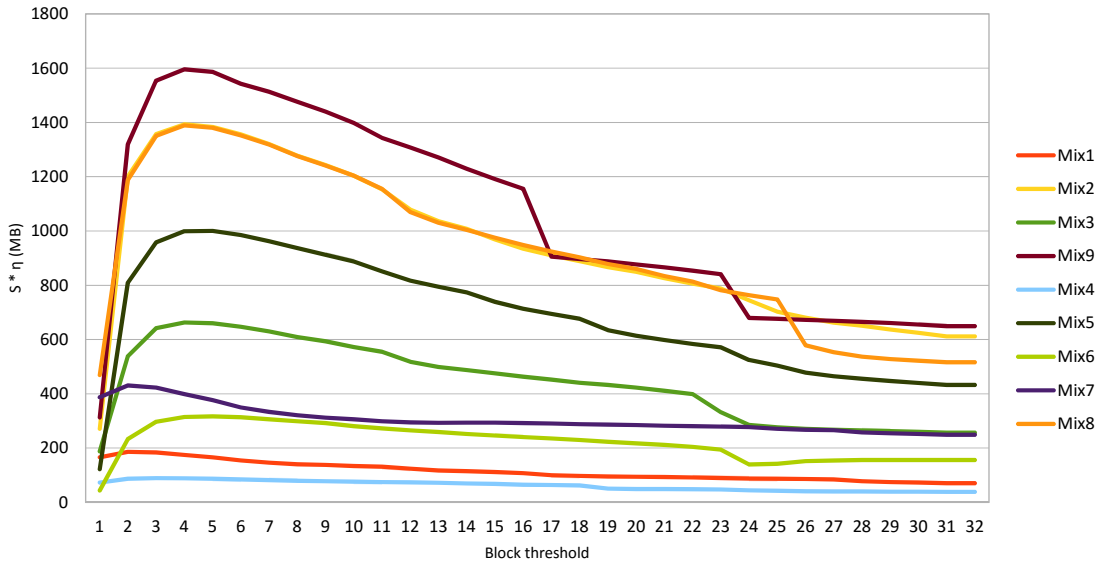


Figure 2.11: The efficiency-savings product for each workload. All workloads show a peak at around 4 blocks.

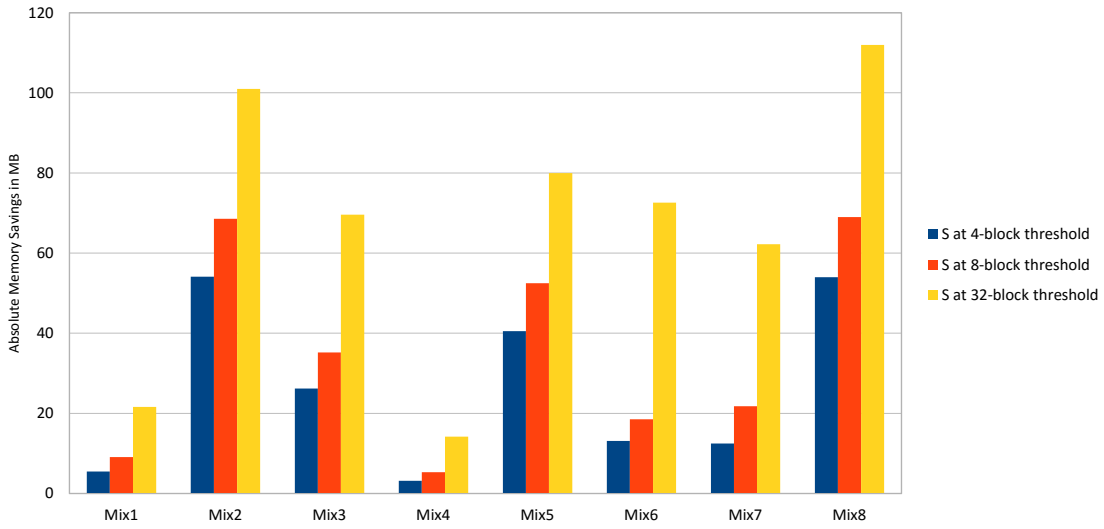


Figure 2.12: The absolute memory savings due to similar page sharing for each workload.

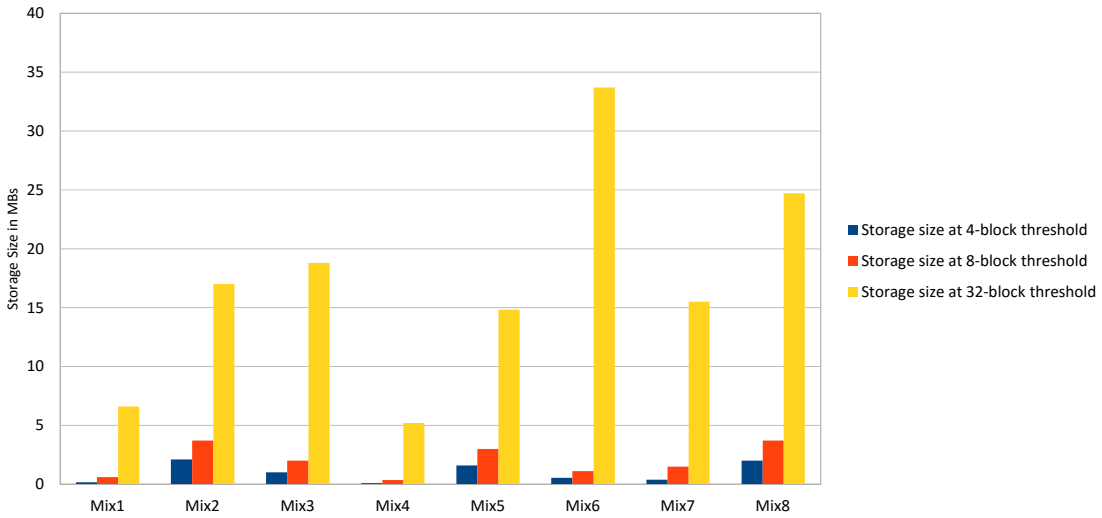
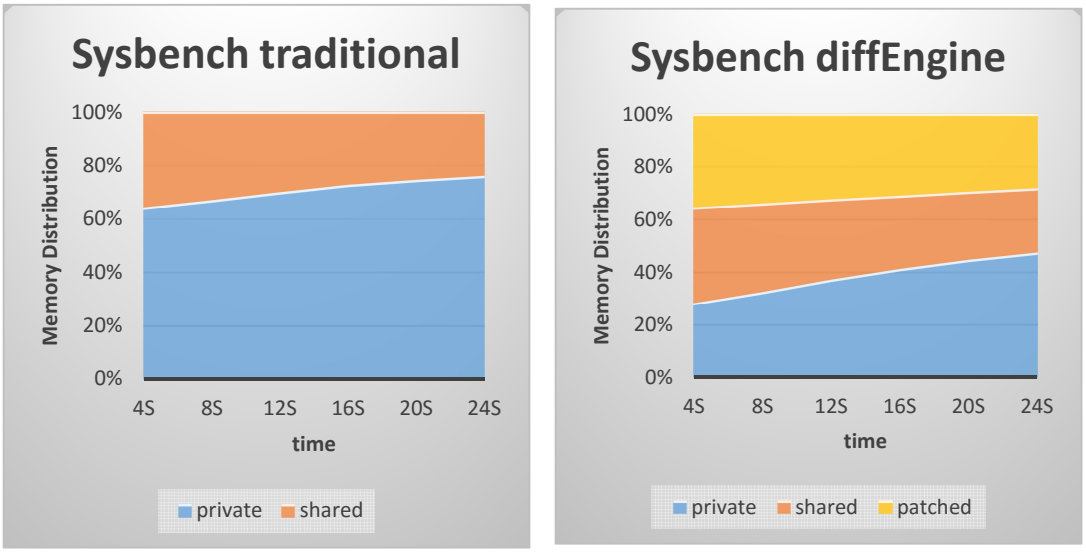
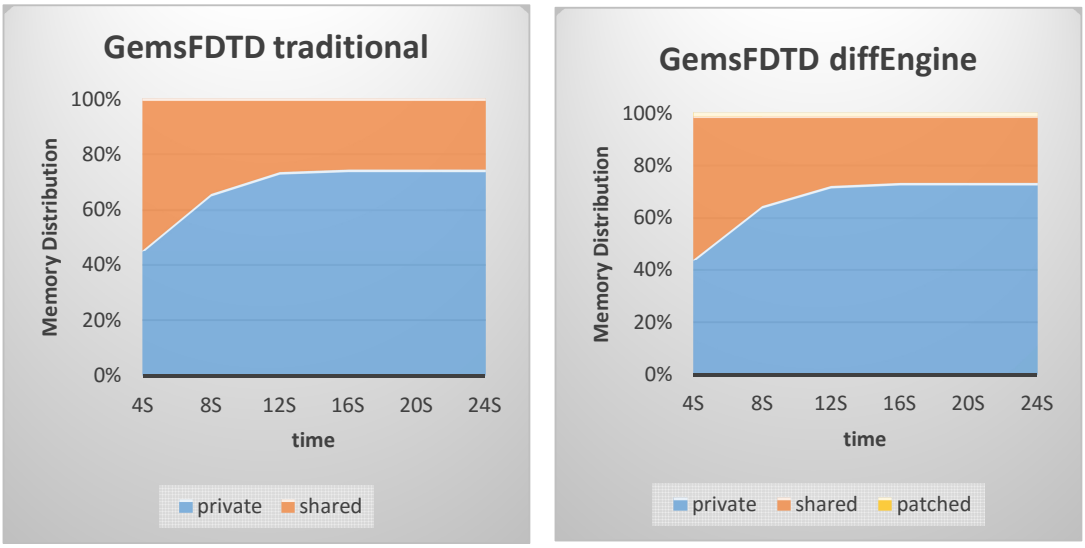


Figure 2.13: The storage size required to achieve the corresponding savings in Figure 2.12.



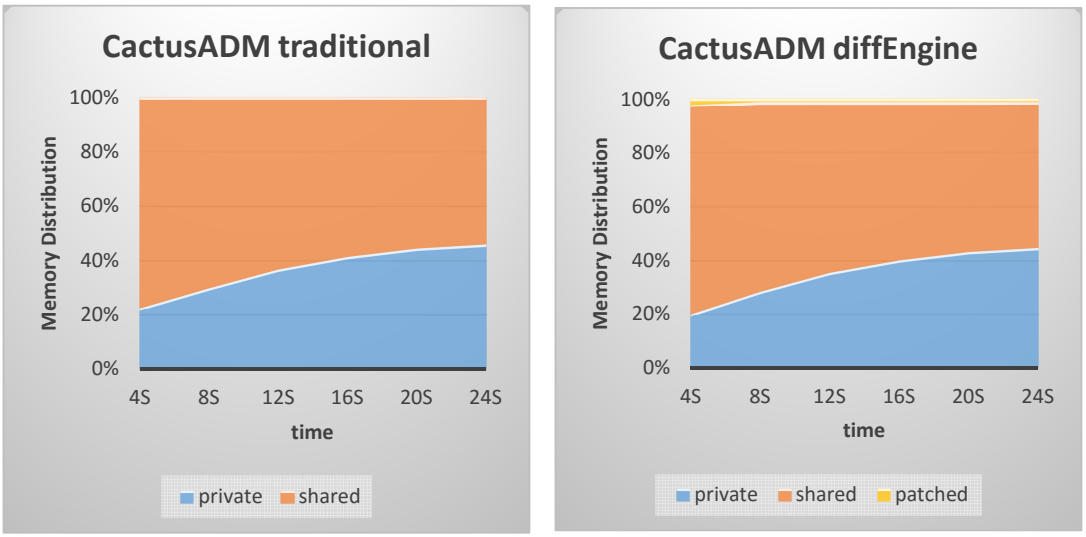
(a) traditional (b) diffEngine

Figure 2.14: Memory distribution of Sysbench Pages



(a) traditional (b) diffEngine

Figure 2.15: Memory distribution of GemsFDTD Pages



(a) traditional (b) diffEngine

Figure 2.16: Memory distribution of CactusADM Pages

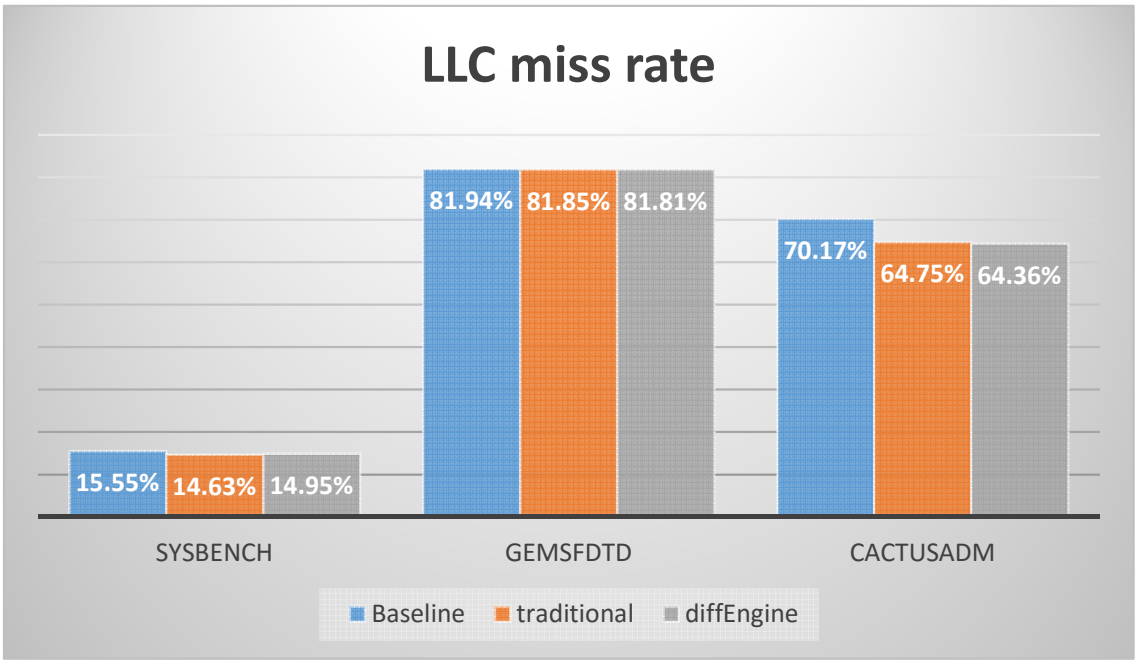


Figure 2.17: The miss rate of the last-level cache (LLC) for each workload

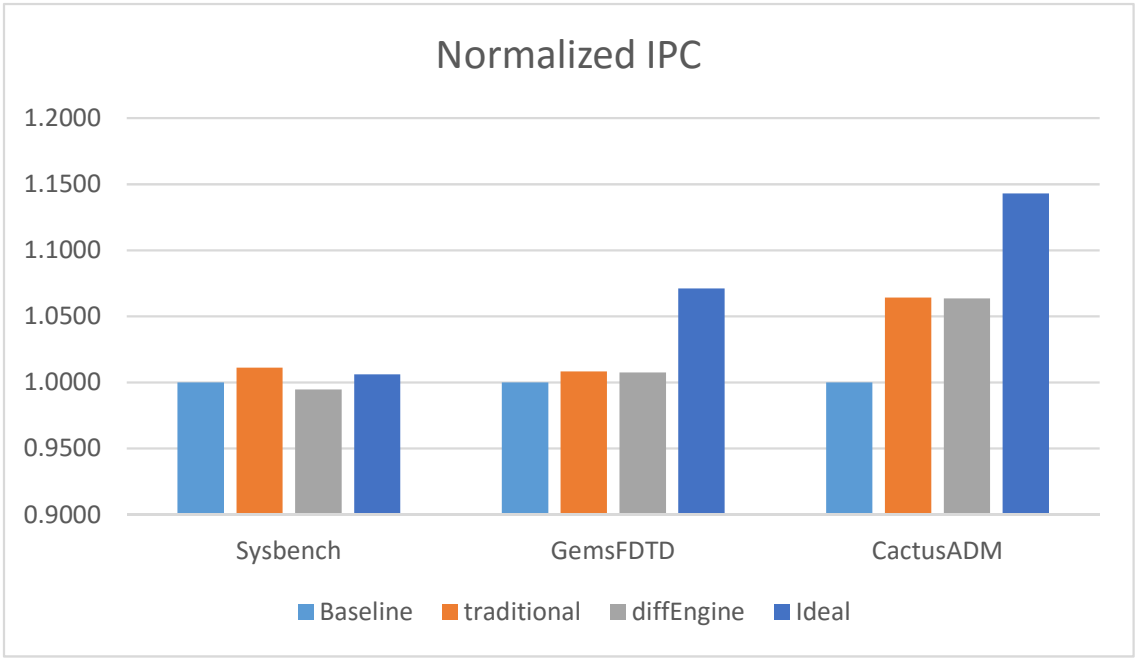


Figure 2.18: Normalized IPC for each workload

Chapter 3

Memory Stability Prediction Framework

> "*Trying to predict the future is like trying to drive down a country road at night with no lights while looking out the back window.*"
>
> — *Peter Drucker*

Prediction has always been an integral part in improving the performance of computer systems. At a lower level, branch prediction techniques [94, 109, 110, 81] are widely used to predict the direction of branches and avoid the performance penalty associated with branches in pipelined processors. As the performance gap between processors and memory increases [34], improving the memory system performance became significantly important through memory access transformation and reordering [47, 67], memory and file prefetching [39, 73, 40, 30, 41, 54, 19, 46], etc. Prefetching techniques are widely used to increase the performance of mem-

ory systems by caching memory blocks that are predicted to be accessed soon avoiding the very expensive cost of accessing data from the main memory.

At a higher level, prediction algorithms [25, 75] were used in software to improve the web performance such as web caching [1, 18, 92], recommendation systems [23, 83], web prefetching [79, 27, 80, 57, 89], and enrich the user experience with search engines [15] and personalized web [84].

In this chapter, we attempt to predict a different aspect of the behavior of the memory that we believe might be useful for many applications. We try to characterize and predict the *stability* behavior of memory pages. By *stability*, we mean the property that the content of a certain memory page remains unchanged for a desirable period of time. This desirable period of time depends on the application or the operation that is using the stability predictor.

We propose a software-based prediction framework that relies on the information provided by the *page flags* that are available through the *Linux* kernel. The Linux *proc* file system provides a lot of information related to the physical memory pages. This information is represented through different flags that are assigned to every page of the main memory. In our characterization, we try to see if there is any correlation between the flags that the Linux kernel assigns to a page and its stability behavior.

Characterizing the stability of memory pages and being able to predict their future behavior can provide some insights that can be useful for various applications. For example, in virtualized environments, both virtual machine live migration and memory deduplication techniques can benefit from an accurate memory page stability prediction. Chapter 4 discusses in detail how such prediction can be effectively used to improve the performance of such applications. We also think that hybrid cache/memory systems can benefit a lot from correctly predicting future writes [2, 3, 60]. Hybrid cache/memory systems are immensely researched as future candidates of caches and main memory systems. This type of memories is designed with both traditional volatile memory like DRAM and non-volatile memory (like STT-RAM and PCM). Predicting stable pages for this type of hybrid memory systems can increase overall performance, cost, and energy efficiency by decreasing memory write latency, write energy, and the write endurance of non-volatile memory cells.

After we characterize the stability of memory pages based on their page flags, we propose a *stability prediction framework* that can be used to predict memory pages that are highly likely to be *relatively stable*. For the rest of the thesis, a *relatively stable* page means that the majority of the page content will remain unchanged during the desired time period. We will give a more thorough definition later in section 3.4 when we introduce our prediction framework.

We propose two classes of prediction: a *conservative* prediction which focuses mainly on prediction accuracy, and an *aggressive* prediction which tries to balance prediction accuracy with prediction coverage. We evaluate and show the results of the proposed prediction framework.

3.1 Introduction

As we mention earlier, a lot of performance can be achieved if we can accurately predict the future behavior of some aspects of hardware and/or software. In this chapter, we focus on trying to predict the stability of a memory page. We believe that an accurate prediction of the stability behavior of memory pages can give insights that can improve the performance of many applications.

For example, in cloud environments where a lot of virtual machines run on many physical hosts, live migration of these virtual machines is a very common operation. *Live migration* of virtual machines is the process of migrating one virtual machine (VM) from a source host to a destination host while the VM is alive and running. The user of the virtual machine should not notice, ideally, any difference while the VM is being migrated. Live migration is an operation that is triggered by the administrator for multiple reasons like online maintenance, fault tolerance, load balancing, etc. One of the most widely used live migration techniques is the *precopy* technique. In precopy, The dirty memory pages are

transferred over several iterations from the source host to the destination host until a maximum number of iterations has been reached or a threshold of dirty pages has been met. Afterwards, the VM is stopped at the source, one further iteration of memory page transmission is performed, and then the VM resumes at the destination. The downside of precopy is the waste of network bandwidth associated with the successive transmission of dirty pages. To alleviate this problem, some compression techniques have been used that tries to send a compressed version of the updates rather than sending the whole page. Having some information about the nature and the stability behavior of the memory pages that are being migrated can result in fewer bytes being transferred over the network. This will be discussed in detail in Chapter 4.

Another very common memory optimization technique in virtualized environments is *memory deduplication*. This is also another example of a technique that can use the knowledge of the future stability behavior of memory pages to achieve better performance. Memory deduplication is a memory optimization technique that tries to reclaim physical memory capacity by identifying duplicate pages and storing only one copy of these pages in the actual machine memory. This is accompanied by marking these merged pages as read only. Further writes to shared memory pages are handled through a copy-on-write exception. When a copy-on-write exception occurs, a new page is allocated for the faulting page. This is

followed by a copy of the full page and then an update to the page table which in turn is accompanied by a TLB shoot down. This means that writes can result in a big performance loss. If we are able to predict the future faulting pages and prevent them from being shared, then the number of copy-on-write exceptions can be reduced. We will also discuss this in detail in Chapter 4.

At a much lower level, we also think that hybrid volatile/non-volatile memory systems may benefit from characterizing and predicting the future stability behavior of memory pages. Non volatile memory systems like PCM are considered to be potential candidates for future cache and/or memory systems. However, these novel non-volatile memories have some inefficiencies. For example, writes are very slow compared to reads, the memory cell can not endure as many writes as a DRAM can, and the energy of writing to these memory cells is very expensive. One solution that has been researched is to have a hybrid memory system where the majority of writes can happen in the DRAM while the *stable* memory pages can reside in PCM or other non-volatile memory technologies. This technique however requires accurately predicting the write accesses [2, 3, 60]. We believe that hardware and software based predictions will be necessary to reach a level of prediction accuracy that is satisfactory for such systems to be viable candidates for future cache and memory systems.

In this chapter, we start by giving an overview of some relevant page flags that we hypothesize to be correlated with the stability of memory pages. We characterize these pages based on their flags and test the validity of our hypotheses.

Based on the characterization, we propose a stability prediction framework that can be used to predict memory pages that are highly likely to be relatively stable. We propose two classes of prediction frameworks: *conservative* prediction which focuses mainly on prediction accuracy, and *aggressive* prediction which tries to balance prediction accuracy with prediction coverage. We evaluate and show the results of the proposed prediction framework.

The rest of the chapter is laid out as follows:

- Section 3.2 motivates our work.
- Section 3.3 discusses the relevant flags that the Linux kernel assigns to pages and characterizes the stability behavior of memory pages based on these flags.
- Based on the characterization, in section 3.4, we propose a pageflag-guided prediction framework that attempts to identify and predict pages that are likely to be relatively stable. That is, the majority of the content of these predicted pages will be unchanged.

- Section 3.5 evaluates the proposed prediction framework and shows the accuracy and coverage of our prediction.

- Finally, section 3.6 concludes this chapter.

3.2 Motivation

As we mentioned earlier, there are various applications and operations that rely on memory page stability for their satisfactory performance. An example of these applications are live migration and memory deduplication.

In memory deduplication, VMware ESX server [103] does not announce if a stability check is performed before sharing identical pages. However, in KSM [4] which is the memory deduplication module in KVM [51], two binary red-black trees are used: A *stable* tree and an *unstable* tree. The purpose of these trees is to track stable and unstable pages respectively. KSM looks into the history of the page to determine if a page is going to remain stable. The *difference engine* [33] also uses a similar method to determine stable pages.

Even though the history of a memory page can give some insightful information about the stability of a page especially in the short term, we believe that such mechanism is not sufficient to predict stability, especially if the desired stability time for the underlying application or operation is relatively high which is the

case for memory deduplication. For example, it is true that pages that have not recently changed are not likely to be written to, yet these pages are also very good candidates for replacement if these pages are not being referenced. To give an example, we show in Figure 3.1 how a significant portion of pages that had stable history for some benchmarks ended up not being stable over a period of 5 minutes.

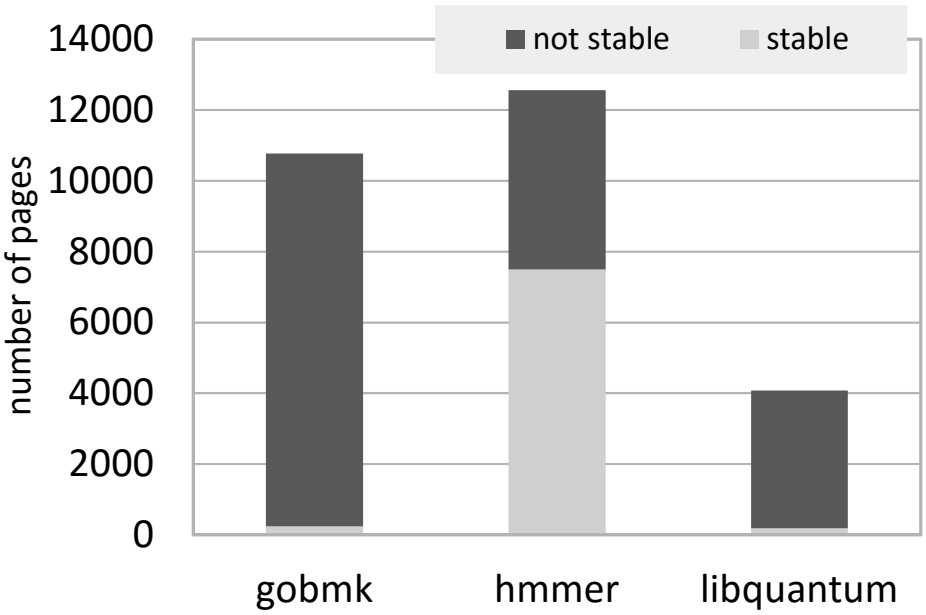


Figure 3.1: The number of pages that were expected to be stable based on their stable history, and the percentage of these pages that *actually* turned out to be stable.

In many cases, the absolute stability of pages is not a strict requirement. Sometimes it is also useful to identify pages that change by a small difference. For example, hardware implementations for fine-grained memory management [91] that are poised to provide hardware support for systems like the *Difference Engine* would benefit significantly from identifying pages that are relatively stable (either stable or those that change by a small difference). In this case, similar pages can be shared allowing shared pages to change dynamically and even rapidly throughout the execution of the program which results in increase memory savings without the performance loss associated with copy-on-write exceptions.

In live migration, [22] tries to detect frequently dirtied pages and postpone the transfer of these pages to the last migration round. Even though this is useful in the traditional way of performing memory live migration to avoid wasting the network bandwidth, current live migration techniques use compression methods that would not adversely impact the network bandwidth only if those frequently dirtied pages are changing by small differences.

For all the above observations, we believe that a prediction framework that can detect if a memory page is likely to be relatively stable can be useful for various applications.

3.3 Pageflag-based Memory Stability Characterization

In this section, we start by giving a brief overview of all the relevant kernel data structures and page flags. Afterwards, we characterize the behavior of memory pages based on the page flags that is available through the Linux kernel.

3.3.1 Kernel data structures and page flags

We start by giving an overview of all the relevant kernel data structure and memory page types.

The Slab Cache

The Linux kernel often needs to allocate memory for kernel data structures and objects. For example, *inodes* and *task structures* are kernel objects that are allocated in the memory by the kernel. These objects are characterized by uniform size and they are frequently allocated and released. In earlier Unix and Linux implementations, the usual mechanisms for creating and releasing these objects were the kmalloc() and kfree() kernel calls.

The performance of kmalloc() and kfree() are not optimized for the small sizes of these kernel objects. One way to solve the inefficiency of allocating these kernel

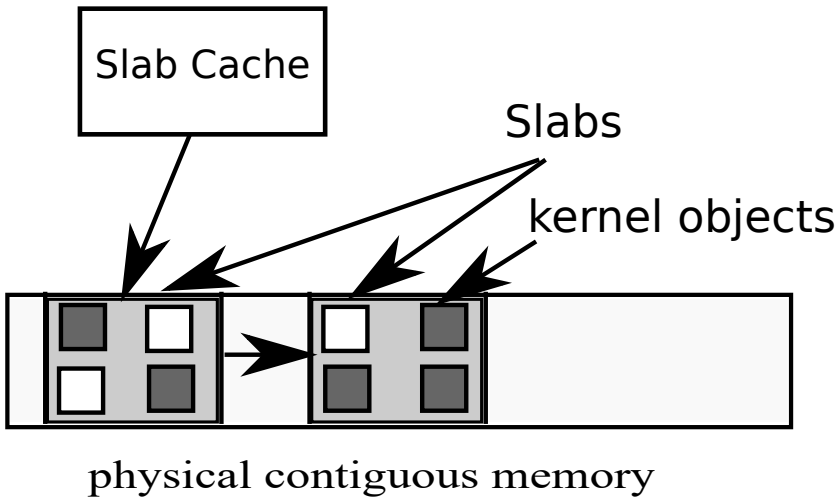


Figure 3.2: The slab cache data structure and the slab pages

objects is to use a kernel cache that is responsible for allocating and releasing these objects. This cache is called the *slab* cache.

Slab memory allocation [13, 12] is a kernel memory management mechanism that allows for an efficient allocation and deallocation of these kernel objects. As shown in Figure 3.2, a *slab cache* is a kernel data structure that manages all the slabs pertaining to a specific kernel object type. A *slab* is a set of one or more contiguous pages of memory set aside by the slab allocator for an individual slab cache. The slab is further divided into equal segments the size of the object that the cache is managing. If a kernel module requires the creation of an object, this object can be fetched directly and efficiently from the preallocated and initialized cache. Destroying an object simply just returns it back to the cache.

Compound Pages

Compound pages are a grouping of contiguous physical pages that can be treated as a single, bigger page. Compound pages can be used to allocate *huge pages*. They are most commonly used by the slab allocator to allocate slabs of higher number of pages.

The first page in a compound page is called the *compound head* page. All the other pages comprising the compound page are called the *compound tail* pages.

Memory Reclamation Policy

Linux uses a least recently used(LRU) replacement policy to replace memory pages that have not been used in a while. For this purpose, the Linux kernel maintains two LRU lists. An *active* LRU list and an *inactive* LRU list. The objective is that the active list should hold the working set of all processes and for the inactive list to hold candidate pages that can be reclaimed and replaced by other pages. New pages start out in the inactive list. If they get referenced, they then get promoted to the active list. Pages get reclaimed when they hit the end of the inactive list.

Memory Mapped Pages

Memory mapped pages are memory pages that are backed by files in the file system. Once a file is memory-mapped, reading or writing to the corresponding memory portion is equivalent to reading from or writing to the file. Memory mapped pages are typically used by process loader in most modern operating systems. When a process starts, the operating system maps the corresponding file to bring the executable segment, associated loadable modules, and shared libraries into memory for execution.

Kernel Page Flags

The *proc* filesystem is a pseudo-filesystem in Unix-like operating systems that presents information about processes and other system information in a hierarchical file-like structure, providing an easy and a convenient method of accessing data held in the kernel. It acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime.

In our work, we are particularly interested in the information that the proc file system can provide regarding the physical pages of the memory. This information can be accessed through:

/proc/kpagecount: This file contains a 64-bit number of the number of times each physical page is referenced, indexed by physical frame number. In other words, how many processes shares the same physical page.

/proc/kpageflag: This file contains a 64-bit set of flags for each page, indexed by physical frame number.

Table 3.1 shows a list of the relevant page flags and their descriptions. The table shows the majority, but not all, of the page flags that the kernel uses. The remaining flags are irrelevant to our discussion. Slab pages will have the slab flag set. A page that is part of a compound page will have either the compound_head or the compound_tail flag set depending on whether the page is the head of the compound page or if it is one of the tails. Any page that is part of the LRU lists will have the LRU flag set. Pages that are in the active LRU list will also have the *active* flag set while those belonging to the inactive list will not have the active flag set. For memory mapped pages, they will have the *mmap* flag set and most of the time, they will also be referenced by more than one process in kpagecount.

3.3.2 Memory Page Stability Characterization

We start this section by trying to characterize the stability behavior of memory pages based on their flags. We assume a typical page size of 4 KB and we divide a memory page into 64 blocks. Each block is 64 bytes.

Table 3.1: Page Flags Description

Flag	*Description*
slab	The page belongs to a slab
compound_head	The page is a compound page head
compound_tail	The page is a compound page tail
LRU	The page is in one of the LRU lists
active	The page is in the active LRU list
mmap	The page is memory-mapped

Given a certain time period τ, we divide the machine memory pages into different classes as follows:

- *Stable* pages are the pages that remain unchanged during τ.
- *Pseudo-stable* pages are the pages that change by less than or equal to 32 blocks, i.e half of the page has changed during τ.
- *Unstable* pages are the pages that change by more than 32 blocks during the time period, even if a machine page is replaced by another page, we refer to this page as *unstable* because the content at that specific machine address range has changed by more than 32 blocks.

Stability hypotheses

We hypothesize that *slab* pages are expected to be stable or pseudo-stable since they hold initialized and frequently used kernel objects. The slab allocator usually assigns slabs that span the size of more than one page associated with one kernel object. Kernel objects are typically characterized by small sizes, and since a free slab slot is already initialized, the difference between a free slot and a busy one is minimal.

Based on our experiments, *compound* pages are primarily used by the slab allocator to allocate slabs that spans the size of more than 2 contiguous pages. Even though this is the case, the *tails* of a compound page will not have the *slab* flag set even if it is part of a slab. Only the *compound_tail* flag will be set. However, the compound page head will have both the *slab* and *compound_head* flags set. For the same reasons a slab page is expected to show relative stability, we also hypothesize that compound pages will exhibit the same behavior.

LRU inactive pages are pages that have not been used in a while and hence these pages are likely to be swapped out to disk which makes these pages unstable for a long enough time period. So we hypothesize that LRU Inactive pages are not good candidates if finding pages that show long-term stability is required (as in memory deduplication). LRU inactive pages will have the *LRU* flag set, but the *active* flag will not be set.

Table 3.2: VM Workloads for testing our hypotheses

Workload	*description*
gobmk	AI game playing benchmark
hmmer	searching a gene sequence database
libquantum	quantum computing benchmark

Another hypothesis we make is about memory mapped files. As we mentioned earlier, most of the memory mapped files are either the process instructions part that is memory mapped from the disk, or they are shared libraries. Pages that belong to shared libraries will, most of the time, have more than one process referencing them in *kpagecount*.

Testing the hypotheses

To test our hypotheses, we run a mixture of one-VM and two-VM workloads over KVM [50]. Each VM is configured with 512 MB of RAM and one vCPU and runs an Ubuntu operating system. Each VM can also run one of the workloads described in Table 3.2. We chose gobmk,hmmer [28], and libquantum from the SPEC CPU2006 benchmarks because they have a different variety of memory footprints. We run these different combinations of one-VM and two-VM workloads, execute one of the benchmarks described in Table3.2, take a snapshot some time after the

execution starts, and check the page flags of all the memory pages. Afterwards, we resume the VMs and check how the memory pages that have flags of interest change over a time period of 1 minute and 5 minutes. We report the cumulative distribution function (CDF) of all pages of interest in Figures 3.3, 3.4, 3.5, and 3.6 for the two different time periods. The x-axis represents the number of 64-byte blocks that have changed.

The results show that slab pages are highly stable, both in the short term and the long term. About 60% of slab pages are stable pages that remain unchanged during the whole time period. The rest of the pages exhibit pseudo-stability, since the majority of the diverging pages change by less than 32 segments. The same behavior is exhibited by compound pages which is intuitive given the strong correlation between slab pages and compound pages.

Memory-mapped pages also show great stability with up to 90% stable pages in the short term and 72% stable pages in the long term. However, unlike slab and compound pages, the majority of diverging pages here are unstable pages with 64 segments of difference. Given the read-only nature of the majority of memory-mapped pages, the results indicate that the instability of the memory-mapped pages arises from them being swapped out to disk and replaced by other pages.

Even though the LRU inactive pages show great stability in the short-term, their stability in the long term is very bad. Only 30% of these pages remain stable

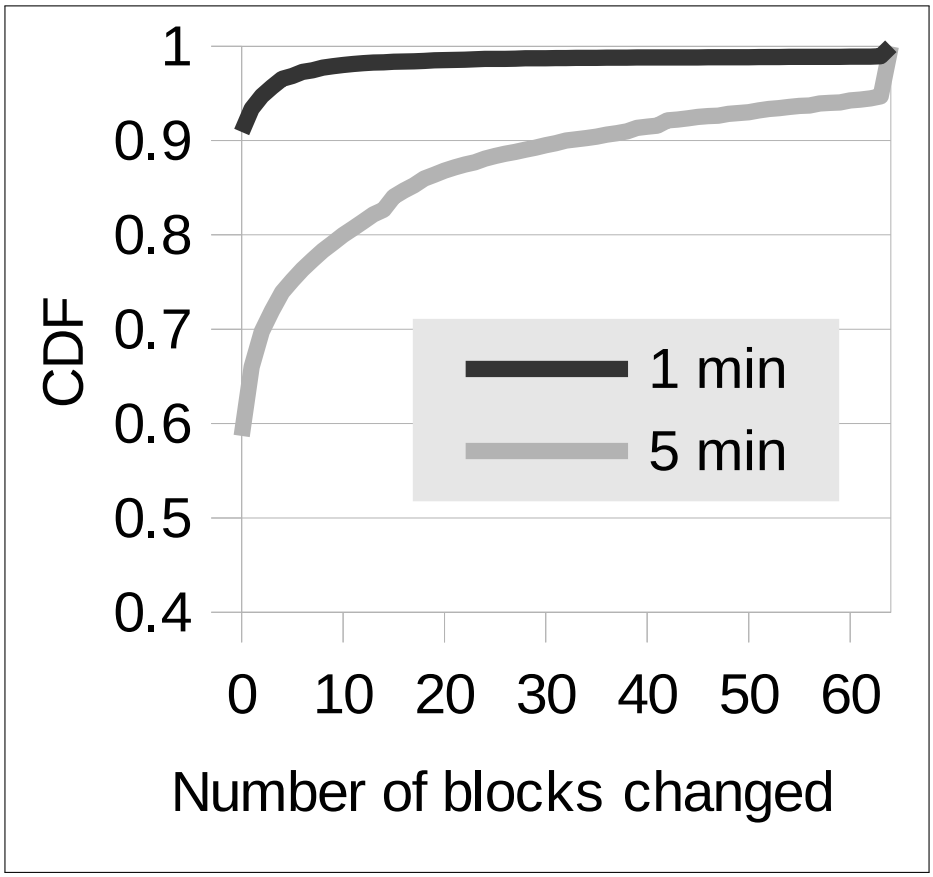


Figure 3.3: Cumulative distribution function of the number of 64-byte blocks that have changed in pages that have the slab flag set during time intervals of 1 minute and 5 minutes

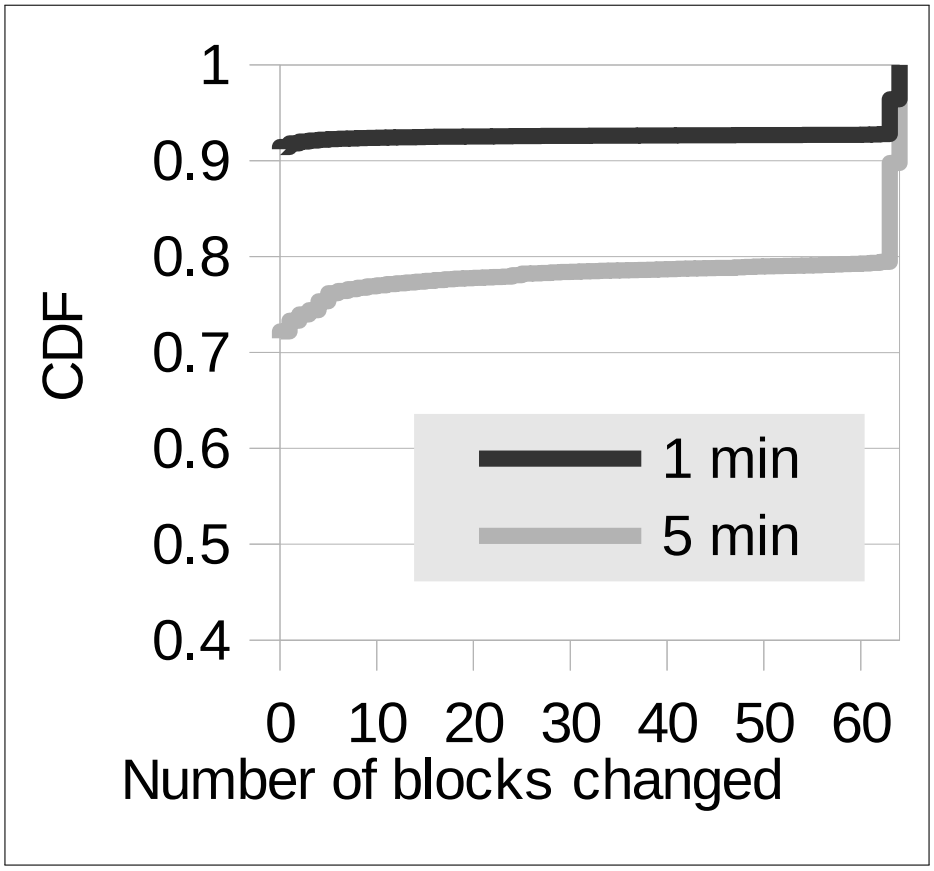


Figure 3.4: Cumulative distribution function of the number of 64-byte blocks that have changed in pages that have the memory-mapped flag set during time intervals of 1 minute and 5 minutes

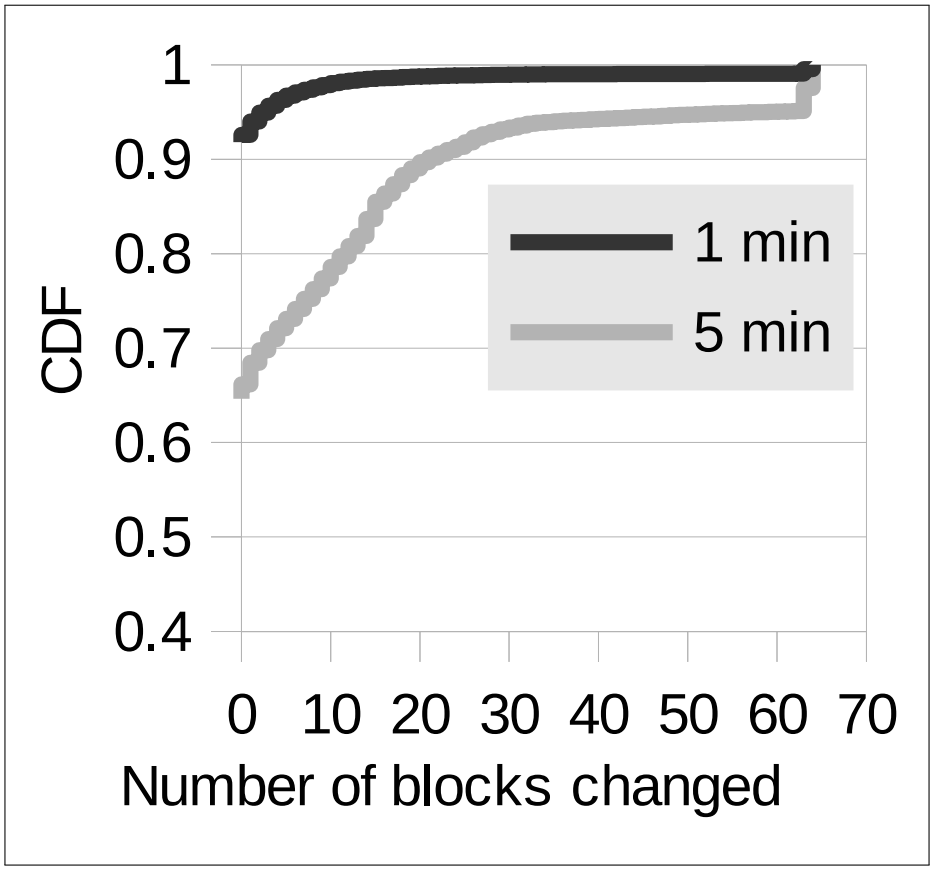


Figure 3.5: Cumulative distribution function of the number of 64-byte blocks that have changed in pages that have the *compound_head* or *compound_tail* flag set during time intervals of 1 minute and 5 minutes

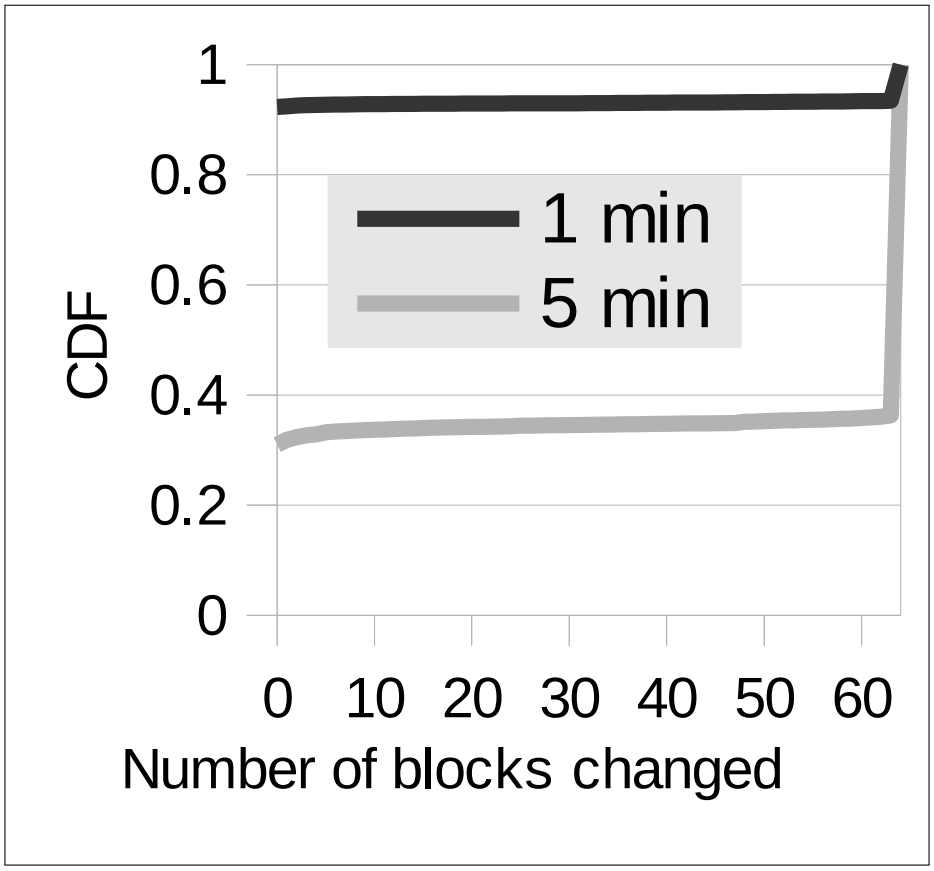


Figure 3.6: Cumulative distribution function of the number of 64-byte blocks that have changed in LRU Inactive pages that have the LRU flag set but not the *Active* flag during time intervals of 1 minute and 5 minutes

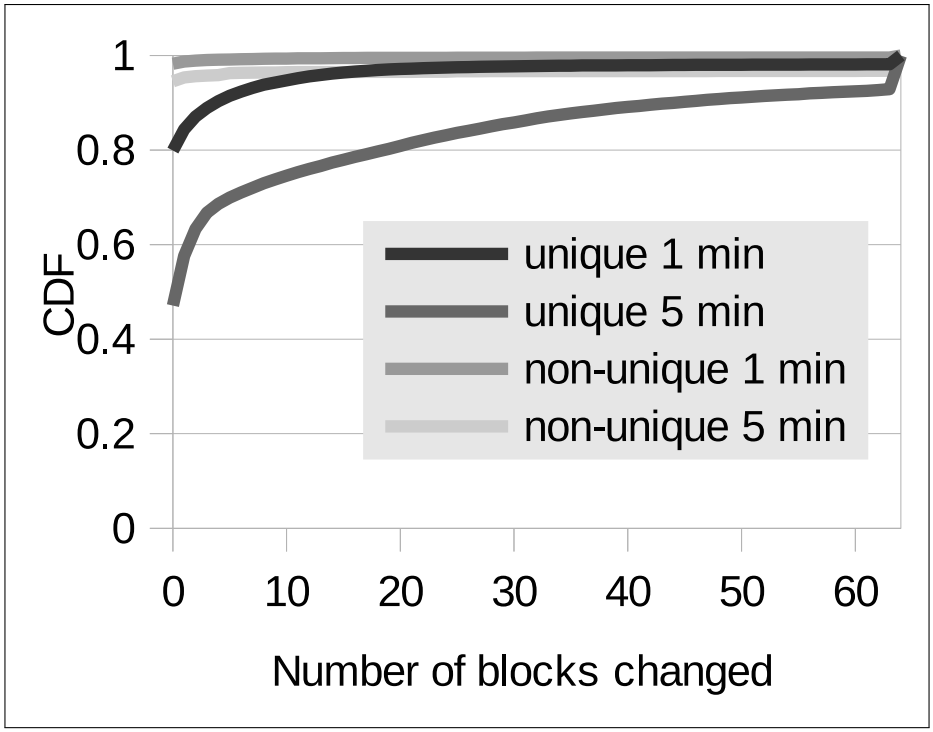


Figure 3.7: Cumulative distribution function of the number of 64-byte blocks that have changed in pages that are unflagged and non-zero during time intervals of 1 minute and 5 minutes. Unique pages are those that are not identical to other pages in other VMs. Non-unique pages are those that are identical to other pages in other VMs.

in a 5-minute interval, with the majority of the diverging pages changing completely by 64 segments. This also indicates that these pages have been swapped out to disk and replaced by other pages, which is an expected outcome.

Since there is not enough information about the actual workload running, it is hard to make an accurate prediction regarding the behavior of LRU pages in the active list. Their behavior is random and is completely dependent on the running workload.

There is a population of pages with no flags associated to them whatsoever. The behavior of this type of pages is also random. However, as we show in Figure 3.7, we noticed that for workloads of two or more virtual machines, If these unflagged pages are non-zero non-unique pages, meaning they are identical to another page in a different VM, then these pages tend to exhibit a very high stability. This stability tends to last even for the long term. Unique non-zero unflagged pages, on the other hand, loses their stability with time. Also, unlike the abrupt transitions in the CDF curves of LRU inactive pages and memory-mapped pages, the smooth CDF curve of these unflagged pages indicates that these pages lose their stability because they are being written to, not because they are swapped out.

3.4 Prediction Framework

Based on our hypotheses and characterization, we propose a prediction framework that can predict memory pages that are likely to be *relatively stable* based on the kernel's page flags. For the rest of the paper, a *relatively stable* page means that a page is either *stable* or *pseudo-stable*.

We divide our prediction framework into two classes: First, *conservative* prediction, which tries to be conservative about prediction. In conservative prediction, we try to avoid all pages that are characterized by an abrupt behavior in their CDF graphs which indicates that these pages are susceptible to being swapped out under heavy memory load. Second, *aggressive* prediction which tries to increase the range of correctly predicted pages relative to the total number of relatively stable pages in the memory, at the risk of some loss in prediction accuracy.

Table 3.3 shows the flags of the pages that we predict to be relatively stable for each of the two prediction classes for the short-term and the long-term. Slab, compound, and non-unique unflagged pages are always safely predicted to be relatively stable since those type of pages have a smooth CDF curve indicating they are pages that are being smoothly written to. They do not have the same abrupt behavior of memory-mapped and LRU inactive pages.

In the aggressive approach, the LRU inactive pages are predicted to be relatively stable, but the same prediction can't hold in the long-term because these

Table 3.3: Page flags associated with conservative and aggressive prediction classes

Class	*conservative*	*aggressive*
short-term	slab, compound, non-unique unflagged	conservative flags, memory-mapped, inactive
long-term	slab, compound, non-unique unflagged	conservative flags, memory-mapped

pages are more likely to be swapped out even if the memory footprint of the applications running in the VM can fit inside the memory. Memory-mapped pages, on the other hand, are predicted to be relatively stable both in the short term and the long term. Even though both LRU inactive pages and memory mapped pages have shown an abrupt diverging behavior, they still have a good percentage of completely *stable* pages which we can take advantage of in our aggressive approach.

3.5 Evaluation and Results

To evaluate the proposed prediction framework, we introduce a set of all possible combinations of two-VM workloads that contains benchmarks as described in Table 3.4. Each benchmark runs on a separate VM with one vCPU and a memory

Table 3.4: Virtual Machine Workloads

Applications	*description*
Apache benchmark	A tool used for benchmarking Apache servers
Sysbench	A database benchmarking utility
Kernel Compile	Compiling the Linux kernel using gcc
Perlbench	Spec CPU2006 version of perl interpreter
Xalancbmk	Spec CPU2006 version of Xalan-C++

of size 512 MB. The *apache benchmark*, *sysbench*, and the *kernel compile* represent the type of workloads that typically run in cloud computing environments and they completely fit within our 512 MB VMs [65, 72].

To test what happens when we run benchmarks that exceeds the available memory size, we included *perlbench* and *xalancbmk*. The large memory footprint of these benchmarks will lead to a large number of swapping. This will negatively affect our prediction accuracy.

There are two performance metrics that we use to measure the effectiveness of the proposed prediction framework. First, prediction *accuracy* which is the ratio of the correct predictions we make to the total sum of predictions we made. Second, prediction *coverage* which is the ratio of the correct predictions we make to the total number of relatively stable pages in the memory. Accuracy measure how

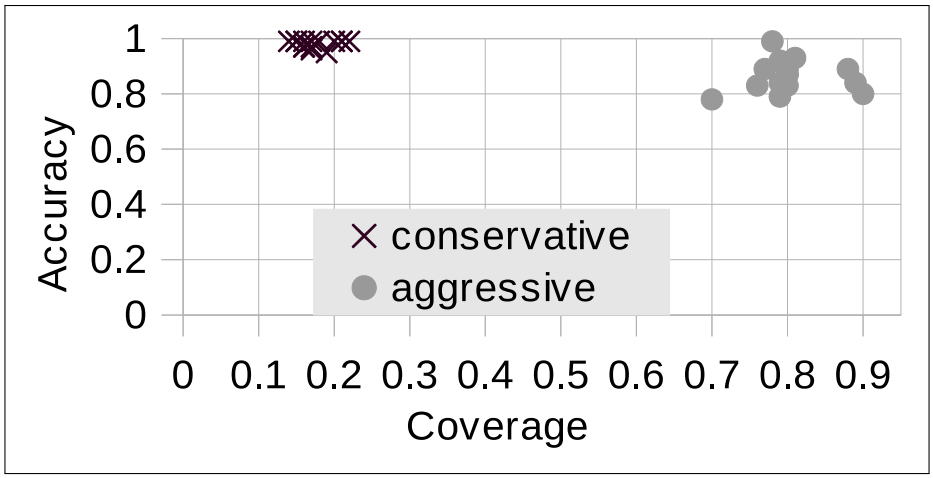


Figure 3.8: Short term accuracy coverage graph of the proposed prediction framework for both the conservative and the aggressive approach.

good our framework is in predicting relatively stable pages. Coverage measures how much of the total relatively stable pages in the memory we are able to predict.

Figure 3.8 and Figure 3.9 show the accuracy-coverage graph of our prediction framework for both the short-term and the long-term respectively. In the short term, conservative prediction results in an accuracy of over 95% for all workloads. However, the conservative approach results in prediction coverage of about 20% for all workloads. The large memory footprint of *perlbench* and *xalancbmk* does not seem to cause any problems in the short run.

In the long run, for workloads that are running benchmarks that can fit within the available VM memory, our prediction accuracy is around 98% and 93% for both the conservative and aggressive methods respectively. The mean coverage for

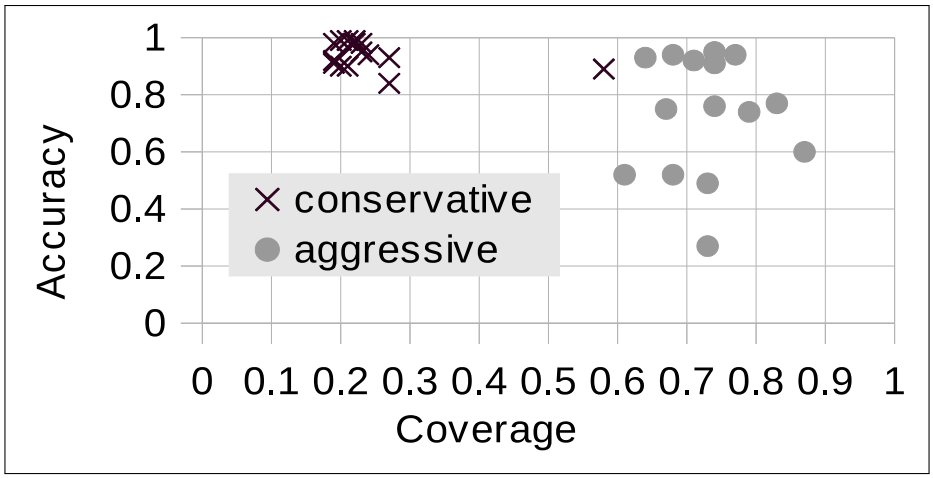


Figure 3.9: Long term accuracy coverage graph of the proposed prediction framework for both the conservative and the aggressive approach.

this type of workloads in the conservative method is 21%. The aggressive method brings this mean coverage up to 71%.

In the conservative approach, a workload of two VMs running *perlbench* results in 89% of prediction accuracy and 58% of prediction coverage. Running *xalancbmk* results in 84% of prediction accuracy and 27% of prediction coverage. The aggressive approach for *perlbench* brings down the accuracy to only 20% but increases the coverage to 73%. While for *xalancbmk*, accuracy declines to 60% and the coverage increases to 87%. All the workloads that has a poor accuracy in the aggressive approach has one or two VMs running either *perlbench* or *xalancbmk*.

Figure 3.10 dissects the percentage of pages that has remained relatively stable and those that has become unstable for each of the relevant flags for both perl-

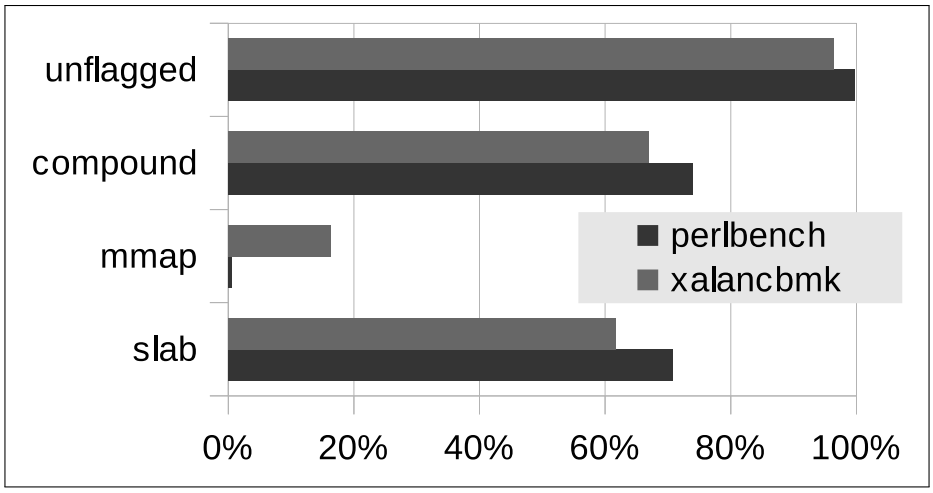


Figure 3.10: Percentage of memory pages with relevant flags that have remained relatively stable for perlbench and xalancbmk

bench and xalancbmk in the long term scenario. The bad behavior of memory mapped pages, resulting from the large memory footprint of these benchmarks, causes the poor accuracy performance in the aggressive approach. It is important to notice that the other flags, and specifically the non-unique unflagged pages are still doing very well even under the large memory load.

3.6 Conclusion

In this chapter, we give an overview of some page flags that can act as good indicators for stability of memory pages. We intuitively hypothesize that slab, compound, and memory-mapped pages should be relatively stable. We test our

hypotheses by tracking samples of pages with various flags and observing how they change in the short term and long term. We also perform memory stability characterization and show that slab, compound, and unique pages lose their instability smoothly and slowly indicating that these pages are pages that are being written to as opposed to memory mapped and LRU inactive pages that lose their instability abruptly indicating they are being completely replaced by other pages.

Based on the above characterization, we propose a prediction framework to predict pages that are relatively stable. We suggest two approaches of prediction: conservative prediction that cares about high accuracy regarding of the percentage of relatively stable pages that are covered, and aggressive prediction that tries to include more pages to increase the coverage at the expense of prediction accuracy. Our results show that for the short term, our conservative and aggressive prediction works very well even for applications whose memory footprint is larger than the available memory for the VM. In the long term, the accuracy of the aggressive approach drops for applications with memory footprint that is larger than the available memory capacity but it works very well for applications that fit within the available memory.

Chapter 4

Applications of the Prediction Framework

> "*There does not exist a category of science to which one can give the name applied science. There are science and the applications of science, bound together as the fruit of the tree which bears it.*"
>
> — *Louis Pasteur*

In the previous chapter, we discussed how we can utilize the page flags to be able to predict the stability behavior of memory pages. We introduced a prediction framework that can predict pages that are likely to be relatively stable. We also introduced two prediction techniques: the conservative technique that tries to prioritize accuracy over coverage, and the aggressive technique that prioritizes coverage. We showed that in the aggressive approach, a significant loss of accuracy

can happen if the memory footprint of the running benchmark is larger than the available VM memory.

Many applications can benefit from our prediction framework. For example, hybrid memory and cache systems that include traditional volatile memory and novel non-volatile memory technologies [2, 3, 60] try to predict memory pages where writes are frequent and store these pages in traditional volatile memories like DRAM or SRAM. The reason is that novel non-volatile memory technologies like PCM and memristors have major inefficiencies in write operations in terms of latency and write energy. Moreover, these cells also have a relatively smaller limit on the number of writes before they were wear out. Our prediction framework can be used to store relatively stable pages in non-volatile memory.

In this chapter, we will discuss two other applications that can benefit from our prediction framework: Memory deduplication and live migration. These two operations are widely used in virtualized environments. The objective of memory deduplication is to reclaim the host machine memory whereas live migration is the process of transferring a running VM from one host to another seamlessly.

4.1 Introduction

A variety of applications can benefit from accurately predicting the stability behavior of memory pages. We explore the benefits of our proposed prediction

framework in two applications that are very popular in the context of virtualized environments.

First, memory deduplication which is a technique used to eliminate memory redundancy between virtual machines running on the same physical host. Often time, these virtual machines are running identical and similar operating systems and/or workloads which result in a lot of memory redundancy. Memory deduplication attempts to reclaim the host machine memory which allows for higher memory utilization and higher consolidation ratio, eventually leading to a better overall performance.

Second, live migration is the process of moving a running VM from one host to another seamlessly without shutting down the VM. In cloud and virtualized environments, operations like online maintenance, load balancing, fault tolerance, and power management are very common. These operations usually entail moving virtual machines around from one host to another.

In this chapter, we study each of these two applications and explore how our prediction framework can be utilized to enhance these operations.

The rest of the chapter is laid out as follows:

- Section 4.2 gives an overview of memory deduplication, discusses the related work, proposes the pageflag-guided memory deduplication, and evaluates and discusses the results of the proposed memory deduplication.

- Section 4.3 gives a brief introduction of VM live migration and the related work. Pageflag-guided live migration is proposed and the results are discussed.

- Section 4.4 concludes this chapter.

4.2 Memory Deduplication

One of the enabling technologies of the pervasive Cloud Computing [5] is Virtualization. In a cloud computing environment, multiple virtual machines (VMs) [31, 88] run on the same physical server. VMs virtualize the whole system allowing for running multiple operating systems with multiple software configurations on the same physical machine. This results in a better server consolidation and fault isolation. A Virtual Machine Monitor (VMM) is a software layer that manages the different virtual machines running on the system and provides resource sharing among these virtual machines.

The cost of the memory system dominates the cost of the whole system and most of the power is consumed in the memory system as well which necessitates an efficient management of the available memory system. It is expected that the cost of an exascale system will be $200M. Half of this cost will be spent on the memory system. Moreover, as the number of cores per chip increases,

more VMs can potentially run on the same physical machine allowing for more server consolidation. However, this increase in the number of VMs running on the same physical machine becomes limited by the system memory capacity. As we increase the number of VMs per node, each VM is allocated a smaller capacity of the memory. We are left with two solutions: First, we can increase the capacity of the system memory. However, Traditional DRAM technology scaling is ending. Also, increasing the number of cores per chip and power density constraints limit the number of DIMMs per node which in turn limit the maximum capacity of memory per node. There is an ongoing research on novel memory technologies that provide higher density and scalability [42, 58, 85] properties than the DRAM. The second solution is to manage the available memory capacity efficiently. There are many ways that were proposed to efficiently manage the available memory capacity. One technique is to use in-memory compression [105] which essentially use compression and decompression techniques to compress pages in memory. A different technique is to use memory deduplication [16, 51, 33, 20]. The basic idea of memory deduplication is to find duplicate memory blocks and only store one copy of these blocks effectively freeing up some redundant memory.

The idea of memory deduplication was preceded by storage deduplication [53, 66]. Storage deduplication was used to reduce bandwidth and storage demands in distributed file systems [74] which is crucial because in distributed file systems,

data blocks are replicated to guarantee reliability and availability. A loss in storage capacity caused by a duplicate data block in this case is multiplicated by the replication factor. Deduplication was also used to reduce the storage demands of VM disk images and checkpoints [44] and backup environments [68].

Memory deduplication allows for:

- Better utilization of memory per virtual machine. For example, all zero pages are reduced to only one zero page in the host memory allowing other virtual machines to use the available memory that is not used.

- Better consolidation since reclaiming memory will allow more virtual machines to run on the same host. This increases the processor utilization and the overall performance of the system.

4.2.1 Background

Memory deduplication is a technique that is used to remove memory redundancy and reclaim some memory capacity. This technique is widely used in virtualized environments. In virtualized environments, it is usually the case that multiple virtual machines that are running on the same physical machine may share the same guest operating system, run similar or identical applications, or work on the same data. All these scenarios create a big possibility that the virtual

machines will have identical memory pages in their memories. Memory deduplication tries to leverage this by storing only one copy of these pages in the host machine memory.

The idea of transparent page sharing was introduced in Disco [16]. The downside of Disco was that it required significant guest OS modifications to identify duplicate pages. Later, VMware ESX server proposed using a hash table to identify memory page duplicates. First, the content of the page is hashed and then the hash table entry is checked for any collisions. If a collision occurs, then a byte by byte comparison is applied to check if the collided pages are identical. If pages are found to be identical, the hypervisor modifies the page table such that these identical pages are mapped to the same machine address.

These merged pages are marked as read only. Writes are handled through a copy-on-write exceptions. When a copy-on-write exception occurs, the hypervisor allocates a new machine page to the faulting page. The content of the page is copied to the new location. These writes cause a performance overhead due to the page fault handling and the TLB shootdown that usually accompanies the page table update.

4.2.2 Related Work

There is a large body of work exploring memory deduplication. We classify prior work into two categories according to the source of memory similarity. The first category is memory deduplication in virtual environments. Independent virtual machines run similar operating systems and applications which result in memory sharing opportunities. Sharing opportunities are sought within the same VM and across different VMs. Typical applications that run on these virtual machines are web servers, database servers, mail servers, etc... The second category is memory deduplication in high performance computing (HPC) environments. Typical applications running on HPC environments are scientific applications. Memory sharing opportunities are sought within the address space of the application itself.

Virtualization-inspired Similarity

Virtual machines on the same physical node often run similar operating systems and similar applications, leading to many duplicate blocks in the machine memory, motivating work on deduplication. Disco [16] first introduced the idea of *transparent page sharing*. Disco shares only specific pages and requires operating system modifications.

VMWare's ESX server [103] improves on this by not requiring operating system modifications, instead using the copy-on-write mechanism. It hashes the contents

and compares pages with identical hash values. This can result in reclaiming 60% of the memory capacity.

The Difference Engine [33] introduced the idea of *similar* pages - pages with *almost* all identical data. They also use memory compression to further increase memory savings. They show that the Difference Engine outperforms VMWare ESX server by a factor of 1.5 for homogeneous workload and by a factor of 1.6-2.5 for heterogeneous workloads. The downside though is the performance overhead. Every time a compressed or patched page is accessed, an exception occurs that is handled by the VMM. As a result, only infrequently accessed similar pages can be merged.

Kernel SamePage Merging (KSM) [4] implements content-based page sharing. Although it was initially developed for KVM [51]. It uses programmer hints to search for merged pages on memory allocation. They have higher comparison overhead. Classification-based memory duplication (CMD) [20] improved this by classifying the pages likely to be similar.

HPC-inspired Similarity

Similar to virtual workload applications, HPC applications are also limited by the system memory. As the problem size increases, a bigger demand on the memory capacity is required. The limits imposed by the number of DIMMs per node

and the significant cost of the memory system, along with power considerations, put a limit to the memory capacity per node. Efficiently managing the memory system is therefore required in HPC systems. In [61], an elaborate study of eight HPC applications running on a Cray XE6 system [100] is presented. Results show that, for HPC applications, there is a significant potential for exploiting memory similarity (identical and almost-identical). Also, the effect of sharing memory pages among processes within the same NUMA domain was studied. Counter-intuitively, increasing the search space to include all processes within the same NUMA domain does not necessarily increase identical and similar pages.

SBLLmalloc [9] is a transparent user-level memory allocation library that intercepts memory allocation requests from MPI applications, automatically identifies identical memory blocks and merges them into one copy using a shared memory object. It limits comparisons to pages of different processes with the same virtual address.

4.2.3 Pageflag-guided Memory Deduplication

In cloud computing environments, multiple VMs typically run on the same physical machine. Most of the time, these VMs run the same operating systems and workloads. This results in a lot of redundancy in the machine memory. Memory deduplication aims at eliminating this redundancy by storing only one

copy of these redundant pages in the machine memory. Detection of duplicate pages is handled by the hypervisor using additional data structures like hash tables [103] or binary trees [4]. After duplicate pages are detected, the hypervisor updates the page table such that these pages share the same physical page in the machine memory. These merged pages are also marked as read only pages. Writes to any merged page is handled through a COW exception. For some workloads, especially homogeneous workloads, memory deduplication can reclaim up to 40% of the machine memory [103].

For memory deduplication, the long-term stability of merged pages is very crucial to both the effectiveness of the memory savings resulting from deduplication, and the performance impact associated with deduplication. From a memory savings perspective, true memory savings are the savings that last long enough until at least following round of memory duplicate detection since these savings can be used for other purposes. Transient memory savings however does not result in any realistic memory savings that can be used for any practical purposes. On the performance impact side, a diverging page causes a COW exception. A COW exception is handled in two steps. First, a new free physical page is identified and then the content of the original page is copied to the new page. Second, the virtual memory page that received the write is remapped to another machine memory page. Both steps incur high latency and are on the critical

path [10, 90, 87, 99, 101]. The copy operation consumes high memory bandwidth [90] and remapping typically requires a TLB shootdown [10, 99]. For these reasons, a better version of copy-on-write has been suggested [91] that handles copy-on-write at a finer granularity.

In our work, we use our prediction framework to predict stable pages and only merge identical pages that were predicted to be stable. After the hypervisor identifies a duplicate page, the flags of the page are checked. If the page is predicted to be stable based on its flags, then this page is a good candidate for merging. If the page is not predicted to be stable, then it is ignored. Both prediction *accuracy* and prediction *coverage* are important for memory deduplication. On one hand, prediction *accuracy* is crucial to avoid the performance penalty associated with the copy-on-write exceptions that occur when a merged page is written to. On the other hand, prediction *coverage* is important for memory savings.

4.2.4 Evaluation and Results

In our experiment, we have different combinations of two VMs running on the same Linux host on KVM. We chose our VM combinations to represent workloads with variable true memory savings and transient memory savings. We take an *initial* snapshot for each VM during the execution of the workload. Afterwards, we identify all identical pages and their respective page flags and page count at

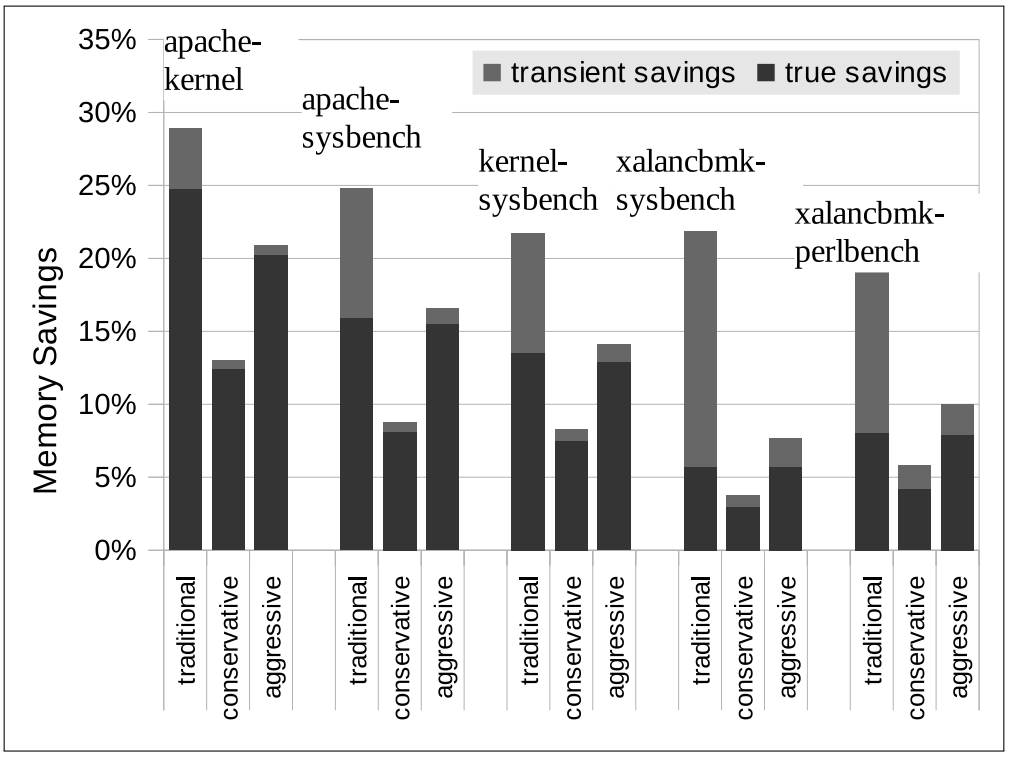


Figure 4.1: True and transient memory savings for each workload for traditional, conservative pageflag-guided, and aggressive pageflag-guided memory deduplication. The figure shows that our aggressive prediction framework results in roughly the same true savings as traditional page sharing without the performance loss associated with diverging pages. The results hold even for benchmarks with memory footprints larger than the VM memory capacity.

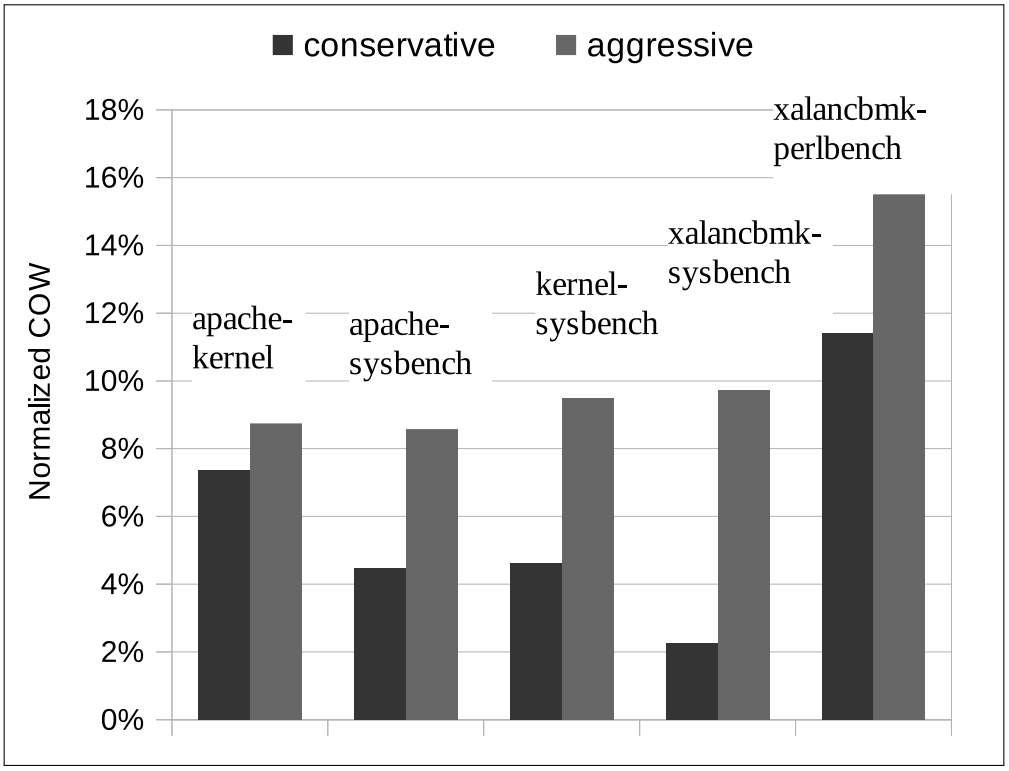


Figure 4.2: The percentage of the number of copy-on-write exceptions occurring throughout the execution of the workload in pageflag-guided memory deduplication relative to that of traditional memory deduplication. The figure shows that our proposed prediction framework can reduce the total number of exception drastically.

this *initial* snapshot. We then run the workload for 5 minutes and record the *true savings* and *transient savings* that occurred during the workload execution. We also record the total number of copy-on-write exceptions that occurred during this period. *Traditional* deduplication presents an upper bound on the memory savings we can get, since it merges all pages regardless of the stability of these pages. This may result in a lot of ineffective transient savings if some of these pages that started out as identical diverged later. That's why traditional deduplication should exhibit the highest number of copy-on-write exceptions. The proposed pageflag-guided deduplication should result in less performance penalty because of the fewer copy-on-write exceptions, depending on our accuracy, at the expense of less memory savings, depending on our coverage.

Figure 4.1 shows the memory savings for each workload for traditional, conservative pageflag-guided, and aggressive pageflag guided memory deduplication. The memory savings shown are the memory savings at the beginning of the workload execution due to sharing identical pages. Not all the savings are significant. The significant portion of the savings, the *true* savings, is the portion that lasts until the end of our run. The other portion, the *transient* savings, are the savings that do not last due to diverging pages. The results show that our prediction framework can help in eliminating most of the transient memory savings without significantly affecting the true memory savings.

Figure 4.2 shows the percentage of the total number of copy-on-write exceptions that occurred during the execution of the workload relative to the exceptions that occurred in traditional deduplication. Our results show that up to 98% reduction of can be achieved using out prediction framework.

4.3 Live Migration

Live migration is the process of moving a VM from a source host to a destination host seamlessly while the VM is running. Many common operations in cloud environments such as fault tolerance, online maintenance, power management, and load balancing require live migration of VMs. In this section, we start by giving an overview of live migration and the related work, then we propose a version of live migration that is guided by our prediction framework and evaluate and discuss our findings.

4.3.1 Background

Virtualization technology allows multiple operating systems to run on the same physical machine providing server consolidation and isolation. Moreover, it is the main enabling technology for cloud computing. In such environments, some operations like online maintenance, load balancing of VMs among the physical

resources, fault tolerance, and power management are quite common for administrators of data centers and clusters. These operations require moving a VM from one physical host to another. The process of moving a VM from a source host to a destination host is called VM migration. Migrating a running VM is called live migration [48]. Live migration entails moving the CPU state, disk state, and the memory state from the source to the destination. The focus of our work is on memory live migration.

The basic idea of live migration in virtual machines was introduced by Clark et. al. [22]. In the paper, memory migration is generally thought of as a process comprising some or all of the following three phases:

Push phase Also called the warm up phase, in this phase, the memory pages are transferred from the source to the destination in the background while the VM is still running at the source. Dirty pages are iteratively resent.

Stop-and-copy phase The VM is stopped at the source host. Dirty pages are transferred for one last time and then the VM starts at the destination.

Pull phase The VM resumes at the destination and any memory page that is requested that is not already at the destination is pulled from the source host.

Based on the above generalized phases, many memory migration approaches have been adopted in research and implemented in hypervisors. The most two common adopted techniques are *Pre Copy* memory migration and *Post Copy* live migration.

Pre-copy Memory Migration

Pre-copy memory migration is the approach introduced by [22]. This approach combines the Push phase and the Stop-and-copy phase. First, the memory pages are transferred from the source to the destination while the VM is running at the source . Afterwards, dirty pages are iteratively resent for multiple iterations. This keeps going for a specific number of iterations or until the number of dirty pages reaches a certain threshold. After this step, the VM is stopped at the source and the remaining dirty pages are copied to the destination followed by the CPU state and registers and then the VM is resumed at the destination. Figure 4.3 shows the precopy live migration timing as depicted by [22]. In stage 0, the VM is running at the source host A. Afterwards in stage 1, the resources on destination host B is reserved. The actual migration starts at stage 2 where the dirty pages of the memory are iteratively copied to the destination. In stage 3, the VM is suspended at the host and a final round of dirty page copying occurs. The last two stages are the stages where the VM finally runs on host B.

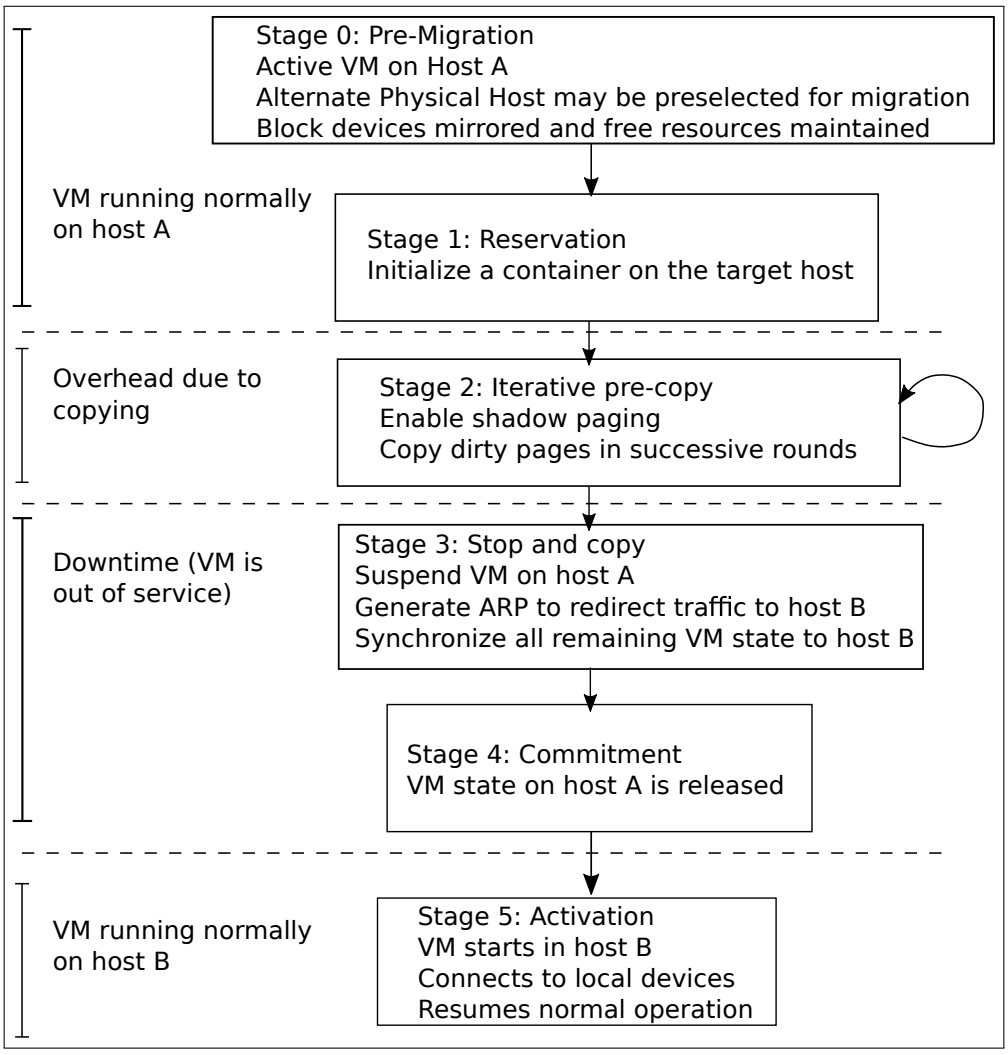


Figure 4.3: VM precopy live migration as depicted by [22].

Pre-copy live migration is the most widely used live migration technique in systems like Xen [6] and Qemu [8]. The premise of pre-copy live migration is to try to minimize the VM downtime by iteratively resending dirty pages until the number of dirty pages reaches a certain threshold or the iteration cap is reached.

One of the advantages of pre-copy is that there is no service degradation other than the VM downtime. Pre-copy does very well when the memory pages are largely unchanged over the duration of migration. However, pages that are written to frequently can significantly impact the performance of pre-copy migration because these dirty pages should be resent over and over again. This wastes valuable network bandwidth, leads to a longer migration time, and increases the VM downtime.

Post-copy Memory Migration

The post-copy memory migration approach [37, 38] adopts the stop-and-copy and the pull phases of the memory migration phases discussed earlier. Post-copy memory migration starts with suspending the VM at the source. Afterwards, the CPU state and registers are transferred to the destination and the VM is resumed. Any memory page that is requested afterwards causes a page fault that requires fetching the required memory page from the source. Also the memory pages can be transferred in the background while the VM is running at the destination.

Post-copy live migration alleviates the shortcomings of pre-copy by immediately suspending the VM at the source, transferring registers and CPU state and resuming execution at the destination. This results in a very small VM downtime and possibly smaller migration time. However, post-copy can result in an unacceptable performance drop due to page faults. If a memory page is requested that is not already at the destination, a page fault occurs and the requested memory page is pulled from the source through the network. Another big drawback of post-copy is that it requires guest OS kernel modifications. The complexity of post-copy prevents it from being widely adopted by common hypervisors.

Figure 4.4 shows the timeline of a post copy live migration to move a VM from host A to host B. In post copy, we first start by a stop and copy phase where the minimal VM state is transferred to host B and the VM is resumed. Afterwards, the page push stage is reached. There are many variants to this stage. As described in [37], the page push stage can have the following variants:

- Demand Paging, this is the slowest of all the variants. Pages are not transferred until they are demanded by the destination. This causes the longest migration time and incurs a significant service degradation

- Active push paging, in addition to demand paging. Pages are also being pushed from the source to the destination.

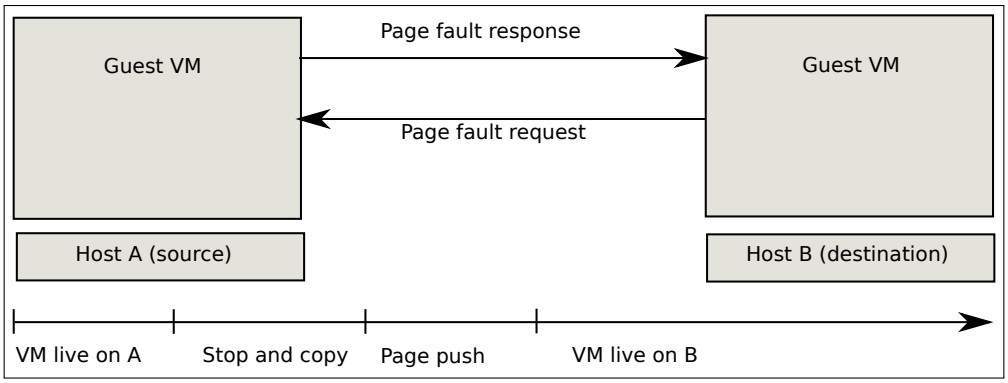


Figure 4.4: VM post copy live migration timeline. A VM is live at host A. If a migration request is initiated to migrate the guest VM from host A to host B, a stop and copy phase takes place where only a minimal VM state is transferred and then the VM starts running on host B. Afterwards, pages are transferred from host A to host B either through demand paging, active push, or prepaging.

- Prepaging, an enhanced active push technique that relies on spatial locality to reduce the number of faulting pages. When a page is demanded, the nearby pages are actively pushed

There is also a variant of live migration which is a hybrid between pre-copy and post-copy. A precopy iteration is performed once before switching to the stop and copy phase. Hybrid migration was first studied in process migration in [77].

4.3.2 Related Work

As we mentioned, the precopy live migration is the most widely used live migration technique due to its simplicity. However, one of the shortcomings of precopy is that it wastes a lot of network throughput due to the iterative transfer of dirty pages. The precopy technique works well if there is a large set of pages that are not updated very frequently because this will minimize the VM down-time. The size of the pages that are frequently updated are called the working set size (WSS). The problem of the WSS in precopy was first discussed by [22]. Some workloads have a small WSS rendering them good candidates for pre-copy migration while others have a large WSS rendering them problematic for pre-copy migration. In [22], pages that has been dirtied since the last iteration are skipped and relegated to the stop and copy phase.

In Svard et. al. [96], the WSS problem is addressed using a combination of techniques. Migration time is reduced through page reordering. Pages of less frequently updated pages have more priority to be transferred over pages with high frequency of updates leaving the frequently updated pages for the last iterations. The VM downtime is reduced by compressing the memory pages that are transferred in the stop-and-copy phase [97]. In Ma et. al., [63], also a bitmap is used to keep track of the frequently updated pages and those pages are sent in the last pre-copy iteration. All these techniques rely on correctly identifying pages that

are not frequently updated by looking into the history of a page. That is, if a page has not been updated for a while, then it likely that this page will remain unchanged.

Another way to solve the WSS problem is to use delta compression techniques to compress the memory pages sent over the network [97, 93]. A compressed version of the memory means that a pre-copy iteration will finish faster allowing for a fewer dirty memory pages to arise between iterations. Delta compression is a compression technique originated from the problem of trying to find the minimum number of edits required to convert one string to another [102]. The compression technique stores data in the form of differences between versions rather than storing everything. In the context of memory migration, a simple bitwise XOR operation between the older and the newer version of the page can be used to generate the delta page. This delta page can be transferred in a compressed form by using a run length encoder (RLE).

However, for the delta compression technique to work. The older version of the dirty page has to be stored in a cache at the source. It is not possible to store all the previous versions of memory pages because this will require a cache size that is equal to the memory size of the virtual machine. Therefore, a cache of smaller size is used to possibly store the pages that are regularly updated. Figure 4.5 shows the delta compression based migration algorithms suggested by [97]. At the source,

each dirt page is checked against the cache. If there is a hit, a compressed version of the delta page is transferred, otherwise, the whole page is transferred. At the destination, if an RLE encoded delta page is received, the original page is decoded and saved.

Qemu adopted the idea of using an XOR based zero run length encoder (XBZRLE) to send a compressed version of the updates of a page instead of sending the whole page completely. This compression needs a cache to store the old contents of the pages being transmitted. In [93], an LRU cache was proposed to hold the older values of memory pages.

4.3.3 Pageflag-guided Live Migration

We propose to use our memory page characterization to anticipate the pseudo-stable pages and give higher preference to these pages to reside in the cache. That's because these pages are predicted to change slowly over the following iterations which in turn translates into a better compression ratio of the cache. Our replacement algorithm depicted in Figure 4.6 tries to combine the advantages of using an LRU cache and the advantages of storing pages that will result in a higher cache compression ratio.

We achieve this through maintaining two LRU lists:

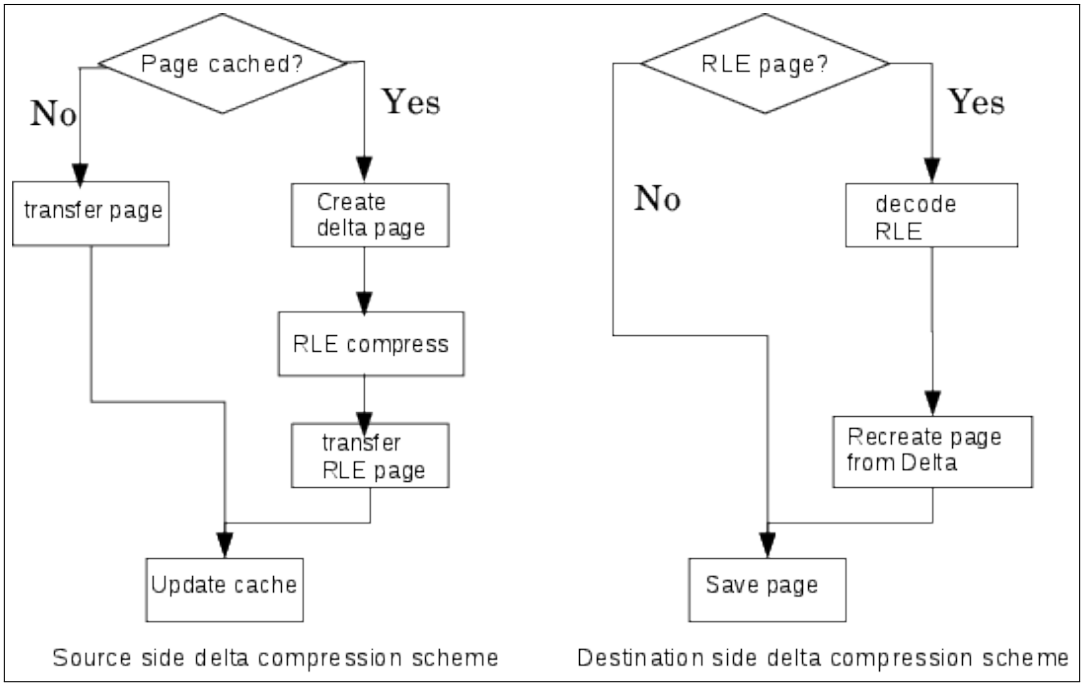


Figure 4.5: The precopy live migration technique that is based on delta compression and run length encoding of memory pages. At the source, each dirt page is checked against the cache. If there is a hit, a compressed version of the delta page is transferred, otherwise, the whole page is transferred. At the destination, if an RLE encoded delta page is received, the original page is decoded and saved [97].

A stable LRU list and an unstable LRU list. The stable LRU list holds all the dirty pages in the cache that were predicted to be relatively stable by our conservative prediction approach. The aggressive approach is not used because the inclusion of memory mapped pages and LRU inactive pages is not useful since the abrupt divergence of these type of pages makes them in either a completely stable state or an unstable state. Both states are not useful to have in the migration cache. Moreover, we include all the unflagged pages, because these pages are also characterized by a smooth divergence curve.

The unstable LRU list holds all the dirty pages that were not predicted to be relatively stable. If the cache is full, any new dirty page attempts to replace the LRU page in the unstable list first before attempting to replace the LRU page in the stable list.

4.3.4 Evaluation and Results

In our experiment, both the source and destination are connected to the same network through a 100 Mbps Ethernet card. We perform a memory live migration for each VM running a workload as described in Table 3.4 using an XBZRLE cache of different sizes. For fair comparison between the traditional LRU replacement policy and our pageflag-guided replacement policy, we continue the live migration until a pre-set total number of dirty pages is reached. We evaluate and discuss the

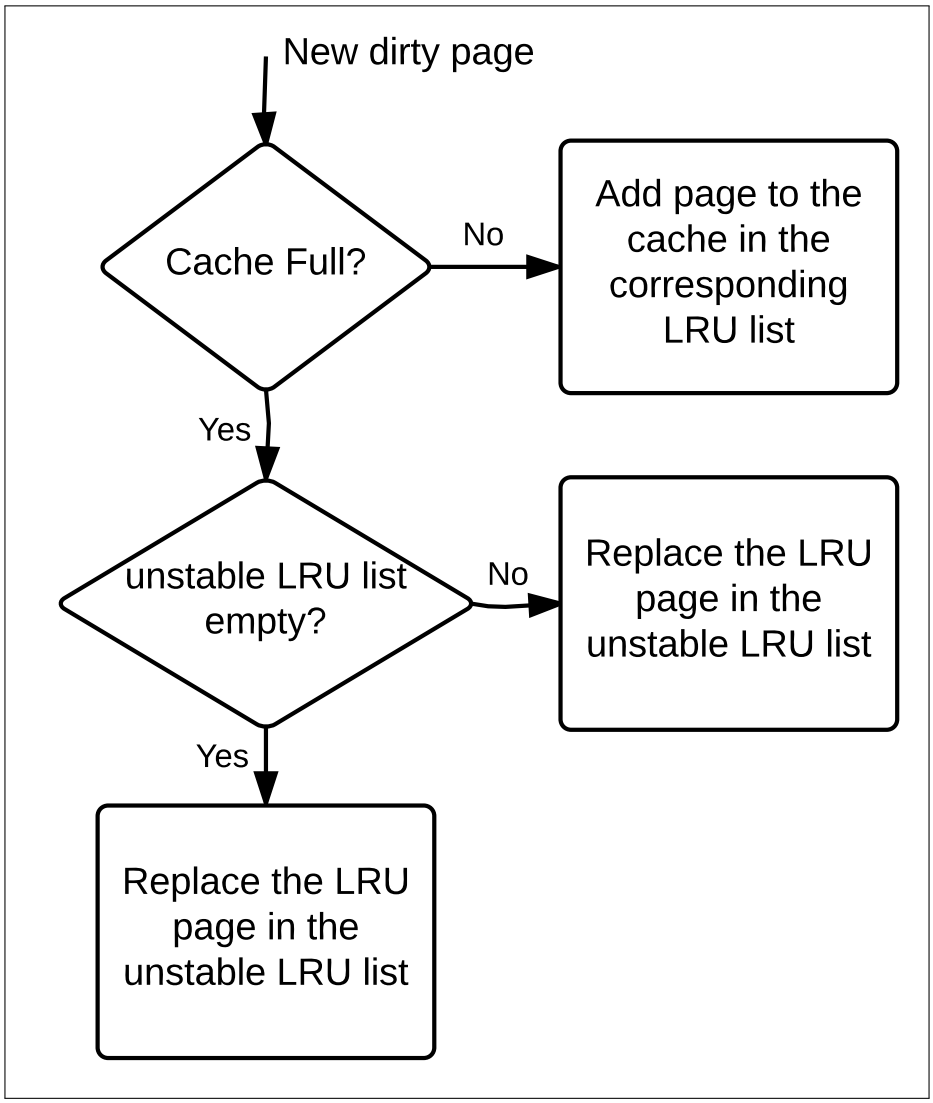


Figure 4.6: The proposed cache replacement policy. Pages that are predicted by our framework to be pseudo-stable are given more preference to reside in the cache than other pages. This is achieved by maintaining two LRU lists. Pages that were not predicted to be pseudo-stable are attempted to be replaced first before replacing pseudo-stable pages.

results of using the pageflag-guided replacement policy to improve the compression ratio of the cache and the number of bytes transferred over the network.

The average compression ratio of the cache is the ratio of the number of bytes transferred over the network for an uncompressed dirty page to the average number of bytes transferred over the network for a page in the occasion that a cache hit occurs. The higher the compression ratio of a cache the better because a higher compression ratio results in a fewer bytes transferred over the network. Figure 4.7 shows the average compression ratio of a 32 MB cache for both replacement policies: the traditional LRU replacement policy, and our proposed pageflag-guided LRU replacement policy. Since our pageflag-guided LRU replacement policy gives preference to pages that are predicted to be pseudo-stable to reside in the cache, depending on the accuracy of our prediction, this should result in a higher compression ratio and fewer bytes transferred. This is especially more pronounced when there is a lot of contention over the finite capacity of the cache. That is, when the number of dirty pages is more than what the capacity of the cache can accommodate. With the exception of *xalancbmk*, our proposed pageflag-guided replacement policy results in a compression ratio improvement that ranges from 8% up to 67%. *xalancbmk* is the only workload where both replacement policies don't work very well. This is because there is a large number of unstable pages

that are changing more frequently than pseudo-stable pages. This fills up the cache with unstable pages.

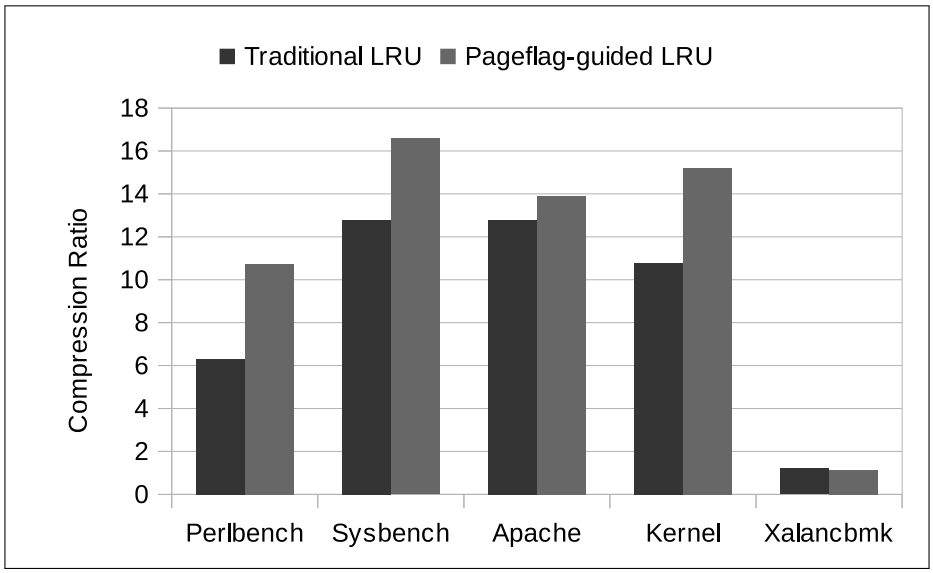


Figure 4.7: Average compression ratio of a 32 MB XBZRLE cache for different workloads. The figure shows that giving higher preference to pages that are predicted to be pseudo-stable increases the average compression ratio of the cache.

Our work mainly aims at decreasing the total number of bytes transferred over the network by increasing the compression ratio of the underlying cache. As mentioned earlier, in *precopy* live migration, there is an initial iteration where the whole memory pages is transferred from the source to the destination. The cache plays no role in this initial phase since all the memory pages will have to be transmitted anyways. Afterwards, only dirty pages are subsequently resent

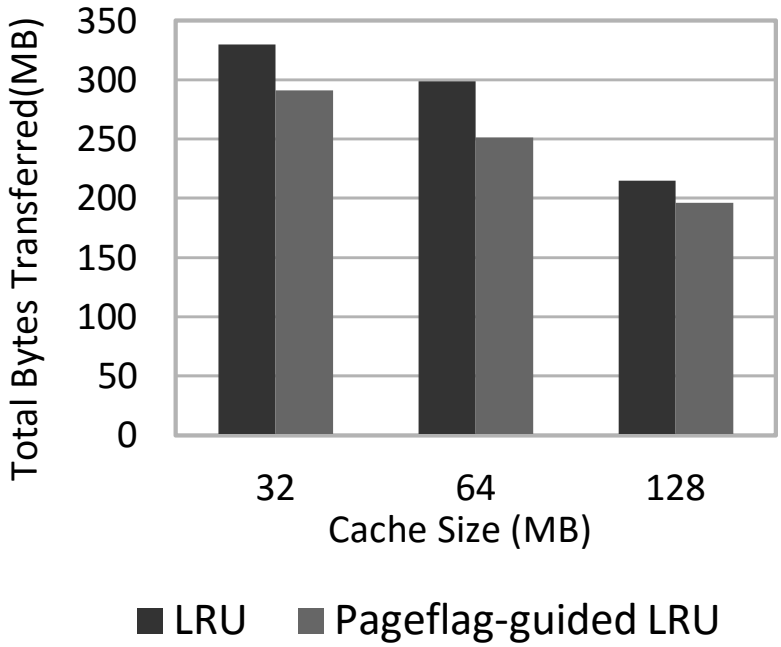


Figure 4.8: The total bytes transferred over the network after the initial iteration while migrating *perlbench* for cache sizes of 32, 64, and 128 MBs

in later iterations. If the old content of a dirty page exists in the cache, then a compressed version of the updates of the page is sent instead of the whole page.

Figures 4.8, 4.9, 4.10, 4.11, and 4.12 show the total number of bytes transferred over the network after the initial iteration for all workloads and three different cache sizes. It is expected that when the WSS of the workload is bigger than the cache capacity, giving preference to pseudo-stable pages to reside in the cache results in fewer transmitted bytes than if unstable pages are residing in the cache. This is attributed to the better compression ratio of pseudo-stable pages. It

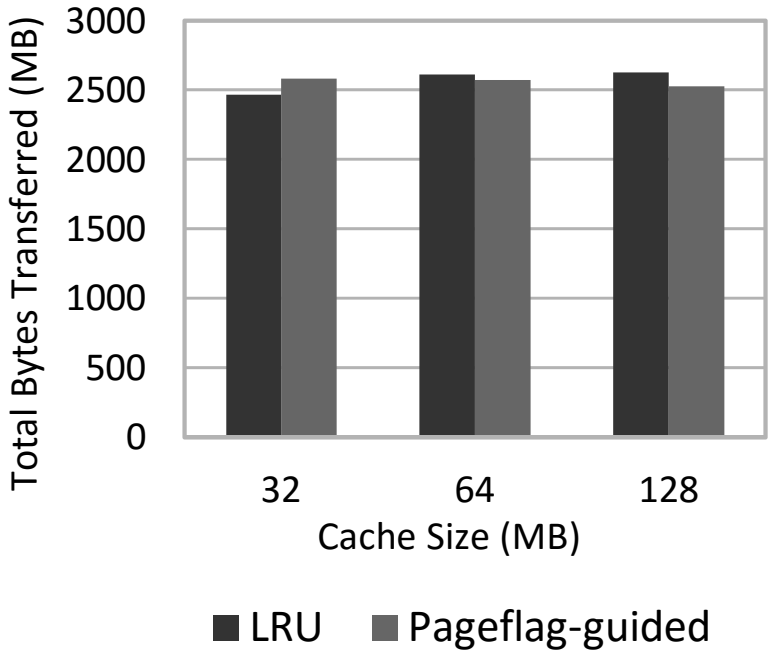


Figure 4.9: The total bytes transferred over the network after the initial iteration while migrating *xalancbmk* for cache sizes of 32, 64, and 128 MBs

is shown that as the cache size increases, the improvement that the pageflag-guided replacement technique achieves decreases. This is because as the cache size increases, both pseudo-stable and unstable pages can be accommodated in the cache. The strength of our proposed replacement policy appears when the available cache size is less than the WSS. In other words, when unstable and pseudo-stable pages are competing for the same finite cache space.

Our results show that our pageflag-guided replacement policy achieves up to 16% decrease in the number of bytes transferred over the network after the initial

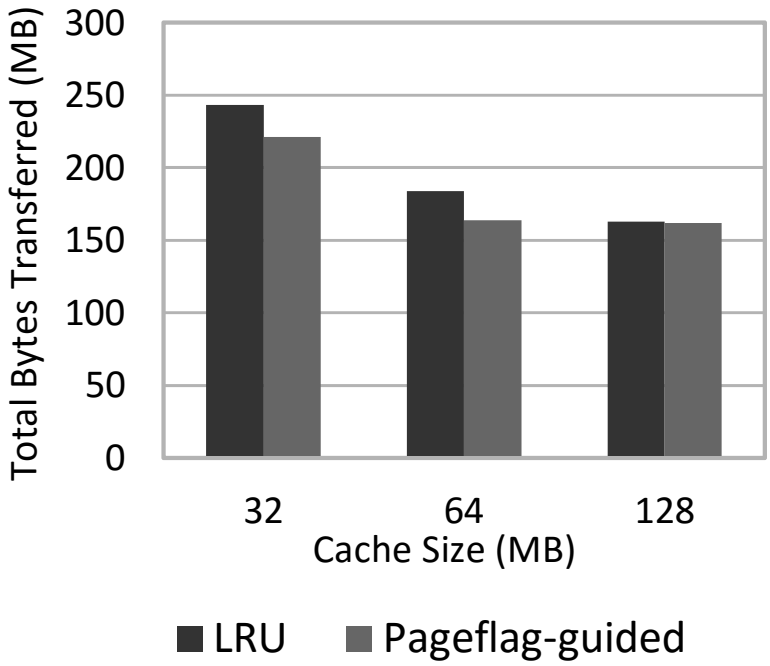


Figure 4.10: The total bytes transferred over the network after the initial iteration while migrating *apache benchmark* for cache sizes of 32, 64, and 128 MBs

transfer of all the memory pages. This is equivalent to about 6% decrease of the total bytes transferred over the network compared to traditional LRU replacement policy. The poor performance in case of Xalancbmk is due to the low short-term prediction accuracy as discussed earlier.

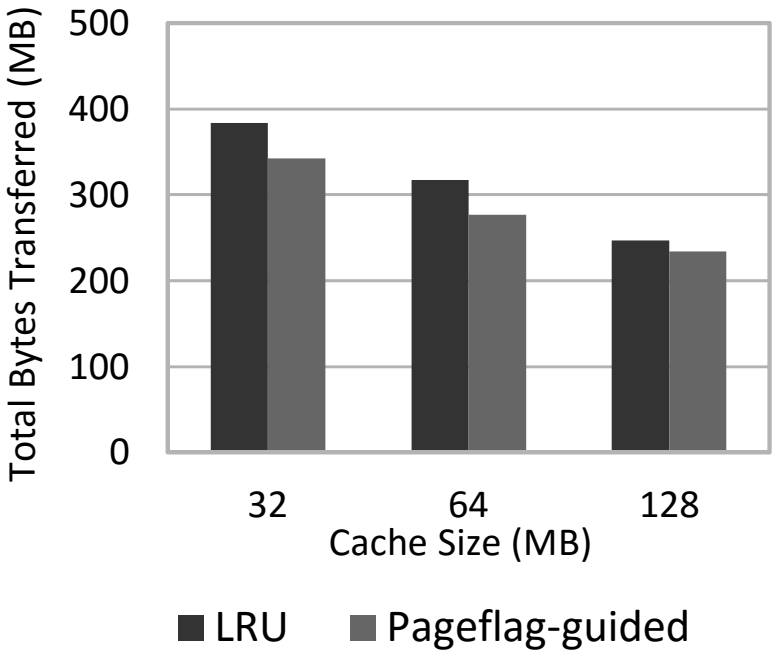


Figure 4.11: The total bytes transferred over the network after the initial iteration while migrating *sysbench* for cache sizes of 32, 64, and 128 MBs

4.4 Conclusion

In this chapter, we discuss how the proposed prediction framework can be used to improve the performance of some applications and operations in the context of virtualized environments.

We start with memory deduplication, a technique that is used in virtualized environments to share memory pages that are identical between virtual machines running on the same physical host. We show that this technique can suffer from a

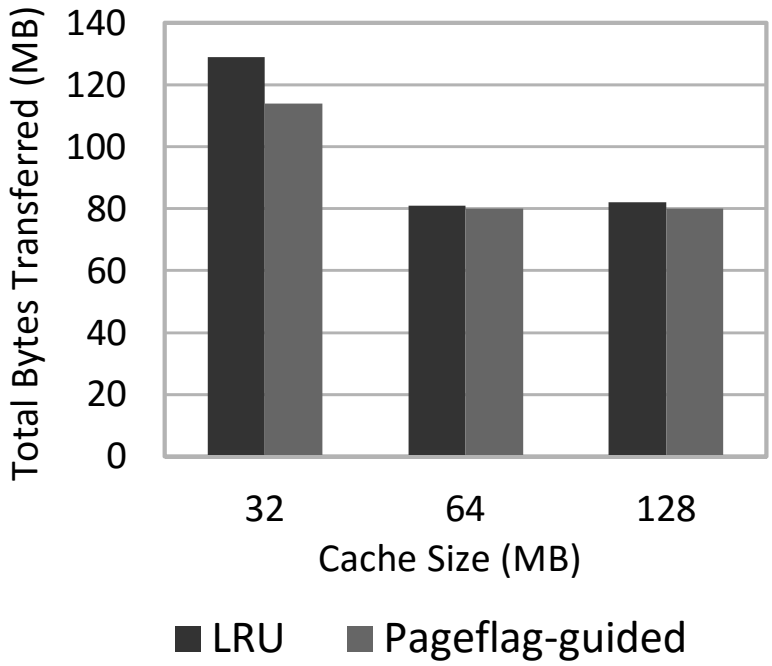


Figure 4.12: The total bytes transferred over the network after the initial iteration while migrating *kernel-compile* for cache sizes of 32, 64, and 128 MBs

performance penalty resulting from the copy-on-write exceptions associated with diverging pages. One way to avoid these exceptions is to accurately predict pages that are stable and only merge those. We use our prediction framework and only merge pages that were predicted to be relatively stable by our framework. Results show that, compared to traditional memory deduplication where everything is shared, we can significantly decrease the number of copy-on-write exceptions without affecting the memory savings.

We also discuss VM live migration as a potential application that can benefit from the proposed prediction framework. Live migration is a very common operation in cloud environments. We show that our prediction framework can be used to increase the efficiency of the underlying cache used through increasing the compression ratio of the pages that hit the cache. Our results show that our pageflag-guided replacement policy achieves up to 16% decrease in the number of bytes transferred over the network after the initial transfer of all the memory pages. This is equivalent to about 6% decrease of the total bytes transferred over the network compared to traditional LRU replacement policy.

Chapter 5

Conclusion and Future Work

In this chapter, we conclude this dissertation and discuss possible future work directions.

5.1 Conclusion

In this dissertation, we start by characterizing the nature of pages that contribute to memory deduplication.

We classify page groups based on their origin into pure and hybrid. We show that the majority of page groups containing identical or similar pages are pure unless the virtual machines are completely homogeneous.

Due to the importance of stability of memory pages contributing to memory deduplication, we also classify pages based on their stability into stable, pseudo-stable, and unstable pages. We show that the majority of pages that start out as identical remain stable, and the identical pages that lose their stability ends

up being completely unstable. This indicates that identical pages do not benefit much from hardware-assisted memory deduplication techniques. On the other hand, the majority of similar pages are stable or pseudo-stable. This indicates similar pages can benefit a lot from hardware assisted deduplication, especially if these pages are frequently read from.

We also show that copy-on-write exceptions associated with software-based memory deduplication techniques can have a non-trivial performance overhead. This underlines the importance of carefully choosing good candidate pages for memory deduplication that have a good relative stability. We also show that memory deduplication can have a good performance side effect which is a better performance of the cache hierarchy.

After we introduce the importance of page stability, we propose a generic prediction framework to predict pages that are relatively stable based on their page flags that is provided by the Linux kernel. We propose two approaches of prediction: conservative prediction that cares about high accuracy regardless of the percentage of relatively stable pages that are covered, and aggressive prediction that tries to include more pages to increase the coverage at the expense of prediction accuracy. Our results show that for the short term, our conservative and aggressive prediction works very well even for applications whose memory footprint is larger than the available memory for the VM. In the long term, the

accuracy of the aggressive approach drops drastically for applications with memory footprint that is larger than the available memory capacity but it works very well for applications that fit within the available memory.

Afterwards, we investigate different applications that can benefit from the proposed prediction framework. We thoroughly discuss how this prediction framework can be used to enhance the performance of memory deduplication by eliminating pages that will eventually diverge. We also discuss how live migration, another common VM operation, can benefit from our prediction framework.

5.2 Future Work

In this section, we discuss possible future work ideas. Our future work ideas are divided into two parts: ideas related to improving the proposed prediction framework, and ideas related to more potential applications that can benefit from the proposed framework.

5.2.1 Stability Prediction Framework

As discussed earlier, looking at the history of the page alone is not enough especially if long term stability is required. Inactive LRU pages have stable history by definition yet they are also very susceptible to be replaced. Combining history-

based prediction with our prediction framework can result in better prediction especially for LRU Active pages. A stable LRU Active page indicates that a page is most likely to be a read-only page that is frequently referenced.

We suggest the following as future work ideas to improve the proposed prediction framework:

1. Our prediction framework can work side by side with history-based prediction. The insights provided through history-based prediction can be useful for predicting the relative stability of LRU active pages.

2. More software hints about memory pages can be utilized for better prediction. For example, what application the page under test belongs to and what the stability history of this application is.

5.2.2 Potential Applications

We discuss three possible future work ideas that can use our proposed prediction framework. First, we discuss the idea of hybrid volatile/non-volatile caches and memory systems. Predicting relatively stable blocks or pages can result in an overall better performance, faster write latency, less write energy, and more memory cell endurance for non volatile memories. Second, we discuss how our prediction framework can be used for hybrid precopy-postcopy migration techniques.

Hybrid Memory Technologies

Persistent storage using hard disk drives relies on mechanical movements and it is a major bottleneck in designing high-performance large-scale systems. DRAM and Flash technologies are used to bridge the latency gap between disks and the rest of the system. However, maintaining the performance growth of such technologies is a challenge because they are hitting a power wall and they are facing physical scalability challenges. Researchers have been looking into other alternative non-volatile memory technologies to sustain this performance growth of the memory hierarchy system [17] such as STT-RAM [104] and PCM [107, 56]. Out of these new memory technologies, phase change memories are coming off as one of the leading and most promising memory technologies. Building a memory system using a phase change material was first discussed in 1960. PCM is based on the hysteresis behavior of chalcogenide glass which can exist in two states:

1. An amorphous state which represents a high resistance state.

2. A crystalline state which represents a low resistance state.

These two states can represent a binary value in a memory cell. Reading is performed by allowing a small current to pass through the cell and measuring its corresponding resistance. Writing to a memory cell means forcing the phase change material to either the crystalline state (SET operation) or the amorphous

state (RESET operation). The SET operation is performed by heating the phase change material above the crystallizing temperature. This is done by passing a moderate current for a long duration across the phase change material. The RESET operation is performed by passing a high current across the phase change material.

The impact that these new non-volatile memory technologies will have on applications require an in-depth understanding of the properties that these new technologies offer and how these new technologies compare to other traditional memory/storage technologies. Table 5.1 [78] shows a comparison between PCM and other technologies regarding performance and density on a 45 nanometer technology. A data-intensive disk-centric application will benefit from the very low read/write latencies of PCMs compared to hard disks without any loss of persistence but at the cost of total capacity, so if the application data can fit into the PCM, PCMs will outperform hard disks. If not, then PCMs can be used as a cache for the hard disk. Although DRAMs are faster than PCM in terms of reading and writing latencies, the non-volatility, the high density, and the scalability of PCMs make it more appealing. In [86], the authors suggest a hybrid main memory system comprised of DRAM and PCM. DRAM offer latency advantages while PCM offers capacity advantages. The advantage of PCMs over Flash is a 500x speedup in terms of read and write latency, a better lifetime, but

less capacity per chip. However, PCMs still hold the scalability advantage over NAND flash.

In [17], the authors project that by 2020, the properties of PCM will be as shown in Table 5.2.

Table 5.1: Comparison between different memory and storage technologies

	DRAM	*PCM*	*NAND Flash*
non-volatility	No	Yes	Yes
idle power	100 mW/GB	≈1 mW/GB	≈ 10 mW/GB
write bandwidth	1 GB/s per die	50 - 100 MB/s per die	5 - 40 MB/s per die
page write latency	2 - 50 ns	≈1 μs	≈500 μs
page read latency	2 - 50 ns	50 ns	≈25 μs
endurance	10^{17}	10^{7}	10^{5}
maximum density	4 Gb	4Gb	64 Gb

These novel non-volatile memory technologies have been studied as replacements to current technologies for cache and memory designs [95, 45, 108, 55, 76, 86, 59]. As we see, one major disadvantage of the non-volatile memory technologies is the write inefficiencies. Writes consume a lot of energy, incurs a lot of latency, and deteriorates the already limited endurance of PCM cells. This is the reason why hybrid cache or memory technologies have been suggested [2, 3, 60].

Table 5.2: Projected PCM characteristics by 2020

capacity	1 TB
Read or Write latency	100 ns
Data rate	$>$ 1 GB/s
Write endurance	10^{12}

The idea is to store relatively stable data blocks in non-volatile memory while blocks that are write intensive can reside in traditional DRAM.

In [2], a hybrid SRAM-STTRAM cache and a write intensity predictor were proposed. The write-intensity predictor correlates write-intense blocks with the address of the memory access instructions.

In [60], a hybrid PCM-DRAM memory system is proposed. Writes are predicted by looking into the write history. Based on the prediction, a new page replacement policy is suggested which tries to store the write intensive pages in the DRAM.

We believe that our prediction framework can work side by side to these predictors and result in more informed decisions for these hybrid cache/memory systems. Using a history-based prediction model combined with some information that the kernel knows about a memory page via page flags can give deeper insights about the stability of a page. These insights can be compiled and used to reach a de-

cision as to where a certain page, or cache block, should reside which may result in a better life span of non-volatile memory cells, faster writes, less energy, and better overall performance.

Hybrid Live Migration

Hybrid live migration is a migration technique that combines pre-copy and post-copy. It is worth noticing that pre-copy is very efficient when the rate of page dirtying is smaller than the throughput of the network, because this means that after some number of iterations the number of dirty pages will be small enough to cause an unnoticeable downtime. Moreover, there is no service degradation because all the memory pages are at the destination when the virtual machine resumes. However, when the working set size(WSS) is large and the dirtying rate of the pages is higher than the network throughput then precopy migration starts to be problematic.

Post-copy solves the WSS problem at the expense of service degradation at the destination when the VM starts working. Combining pre-copy and post-copy may result in a better live migration algorithm. However, for such technique to be effective, only pages that are relatively stable should be precopied. Unstable pages should be postcopied.

Our short-term aggressive prediction framework can be used to predict these pages that are relatively stable. The remaining pages can be post-copied using any of the discussed post-copy variants. Also, we believe that our prediction framework can work side by side with other history-based prediction frameworks to maximize the number of pages that are precopied and in return decrease the service degradation associated with post-copy.

Bibliography

[1] Charu Aggarwal, Joel L Wolf, and Philip S Yu. Caching on the world wide web. *Knowledge and Data Engineering, IEEE Transactions on*, 11:94–107, 1999.

[2] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi. Write intensity prediction for energy-efficient non-volatile caches. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, pages 223–228. IEEE Press, 2013.

[3] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi. Dasca: Dead write prediction assisted stt-ram cache architecture. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 25–36. IEEE, 2014.

[4] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. *Proceedings of the linux symposium*, pages 19–28, 2009.

[5] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing, 2010.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[7] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. An empirical study of memory sharing in virtual machines. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 273–284, 2012.

[8] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[9] Susmit Biswas, Bronis R. de Supinski, Martin Schulz, Diana Franklin, Timothy Sherwood, and Frederic T. Chong. Exploiting Data Similarity to Reduce Memory Footprints. *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 152–163, 2011.

[10] David L Black, Richard F Rashid, David B Golub, and Charles R Hill. *Translation lookaside buffer consistency: a software approach*, volume 17. ACM, 1989.

[11] Daniel G Bobrow, Jerry D Burchfiel, Daniel L Murphy, and Raymond S Tomlinson. Tenex, a paged time sharing system for the pdp-10. *Communications of the ACM*, 15(3):135–143, 1972.

[12] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many cpus and arbitrary resources. In *USENIX Annual Technical Conference, General Track*, pages 15–33, 2001.

[13] Jeff Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[14] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.

[15] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine, 1998. In *Proceedings of the Seventh World Wide Web Conference*, 2007.

[16] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors, 1997.

[17] Geoffrey W Burr, Bülent N Kurdi, J Campbell Scott, Chung Hon Lam, Kailash Gopalakrishnan, and Rohit S Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464, 2008.

[18] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[19] Mark J Charney and Anthony P Reeves. Generalized correlation-based hardware prefetching. Technical report, Technical Report EE-CEG-95-1, Cornell University, 1995.

[20] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. CMD: Classification-based Memory Deduplication through Page Access Characteristics. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2014.

[21] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P Stevenson, and Omid Azizi. Hicamp: architectural support for efficient concurrency-safe shared structured data access. In *ACM SIGPLAN Notices*, volume 47, pages 287–300. ACM, 2012.

[22] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI'05 Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, 2005.

[23] Jeffrey Dean and Monika R Henzinger. Finding related pages in the world wide web. *Computer networks*, 31:1467–1479, 1999.

[24] Peter J Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[25] Mukund Deshpande and George Karypis. Selective markov models for predicting web page accesses. *ACM Transactions on Internet Technology (TOIT)*, 4:163–184, 2004.

[26] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 74–85. IEEE Computer Society, 2005.

[27] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson. Web prefetching between low-bandwidth clients and proxies: potential and performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 27, pages 178–187. ACM, 1999.

[28] Robert D Finn, Jody Clements, and Sean R Eddy. Hmmer web server: interactive sequence similarity searching. *Nucleic acids research*, page gkr367, 2011.

[29] John Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, 1961.

[30] Jason Fritts. Multi-level memory prefetching for media and stream processing. In *Multimedia and Expo, 2002. ICME'02. Proceedings. 2002 IEEE International Conference on*, volume 2, pages 101–104. IEEE, 2002.

[31] Robert P. Goldberg. Survey of virtual machine research, 1974.

[32] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[33] Diwaker Gupta, Sangmin Lee, and Michael Vrable. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53:309–322, 2010.

[34] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[35] Val Henson. An analysis of compare-by-hash. In *HotOS*, pages 13–18, 2003.

[36] Val Henson, Richard Henderson, et al. Guidelines for using compare-by-hash, 2005.

[37] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.*, 43(3):14–26, July 2009.

[38] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[39] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408. IEEE Computer Society, 2006.

[40] Ibrahim Hur and Calvin Lin. Feedback mechanisms for improving probabilistic memory prefetching. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 443–454. IEEE, 2009.

[41] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 247–259. ACM, 2013.

[42] Joe Jedelloh and Brent Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *IEEE Symposium on VLSI technology (VLSIT)*, 2012.

[43] Lei Jiang, Youtao Zhang, and Jun Yang. Mitigating write disturbance in super-dense phase change memories. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 216–227. IEEE, 2014.

[44] Keren Jin and Ethan L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, 2009.

[45] Adwait Jog, Asit K Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R Das. Cache revive: architecting volatile sttram caches for enhanced performance in cmps. In *Proceedings of the 49th Annual Design Automation Conference*, pages 243–252. ACM, 2012.

[46] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 252–263. ACM, 1997.

[47] Mahmut Kandemir, J Ramanujam, and Alok Choudhary. Improving cache locality by a combination of loop and data transformations. *Computers, IEEE Transactions on*, 48(2):159–167, 1999.

[48] Divya Kapil, Emmanuel S Pilli, and Ramesh C Joshi. Live virtual machine migration techniques: Survey and research challenges. In *Proceedings of the 2013 3rd IEEE International Advance Computing Conference, IACC 2013*, pages 963–969, 2013.

[49] Tom Kilburn, David BG Edwards, MJ Lanigan, and Frank H Sumner. Onelevel storage system. *Electronic Computers, IRE Transactions on*, (2):223–235, 1962.

[50] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[51] Avi Kivity, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[52] Jacob Faber Kloster, Jesper Kristensen, Arne Mejlholm, and Gerd Behrmann. On the feasibility of memory sharing: Content-based page sharing in the xen virtual machine monitor. Master's thesis, Aalborg University, 2006.

[53] Ricardo Koller and Raju Rangaswami. I/O deduplication: utilizing content similarity to improve I/O performance. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, pages 211–224, 2010.

[54] Tom M Kroeger and Darrell DE Long. Design and implementation of a predictive file prefetching algorithm. In *USENIX Annual Technical Conference, General Track*, pages 105–118, 2001.

[55] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 256–267. IEEE, 2013.

[56] Stefan Lai. Current status of the phase change memory and its future. In *Electron Devices Meeting, 2003. IEDM'03 Technical Digest. IEEE International*, pages 10–1. IEEE, 2003.

[57] Bin Lan, Stephane Bressan, Beng Chin Ooi, and YC Tay. Making web servers pushier. In *Web Usage Analysis and User Profiling*, pages 112–125. Springer, 2000.

[58] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative, 2009.

[59] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[60] Soyoon Lee, Hyokyung Bahn, and Sam H Noh. Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures. *Computers, IEEE Transactions on*, 63(9):2187–2200, 2014.

[61] Scott Levy, Patrick G. Bridges, Kurt B. Ferreira, Aidan P. Thompson, and Christian Trott. Evaluating the feasibility of using memory content similarity to improve system resilience. In *ROSS '13 Proceedings of the 3rd*

International Workshop on Runtime and Operating Systems for Supercomputers, 2013.

[62] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[63] Fei Ma, Feng Liu, and Zhen Liu. Live virtual machine migration based on improved pre-copy approach. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, pages 230–233, July 2010.

[64] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[65] Vikram Makhija, Bruce Herndon, Paula Smith, Lisa Roderick, Eric Zamost, and Jennifer Anderson. Vmmark: A scalable benchmark for virtualized systems. *VMware Inc, CA, Tech. Rep. VMware-TR-2006-002*, 2006.

[66] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. *Proceedings of the ACMIFIPUSENIX international middleware conference companion on Middleware 08 Companion Companion 08*, pages 12–17, 2008.

[67] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[68] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference on - SYSTOR '09*, page 1, 2009.

[69] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Ksm++: Using i/o-based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, 2012.

[70] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Xlh: More effective memory deduplication scanners through cross-layer hints. In *USENIX Annual Technical Conference*, pages 279–290, 2013.

[71] Grzegorz Miłós, Derek G Murray, Steven Hand, and Michael A Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 1–1, 2009.

[72] Kim-Thomas Moller. *Virtual machine benchmarking*. PhD thesis, 2007.

[73] Jaime H Moreno, Jude A Rivers, and John-David Wellman. Method and apparatus for memory prefetching based on intra-page usage history, January 13 2004. US Patent 6,678,795.

[74] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system, 2001.

[75] Alexandros Nanopoulos, Dimitris Katsaros, and Yannis Manolopoulos. Effective prediction of web-user accesses: A data mining approach. In *Proceedings of the WEBKDD Workshop*, 2001.

[76] Anurag Nigam, Clinton W Smullen IV, Vidyabhushan Mohan, Eugene Chen, Sudhanva Gurumurthi, and Mircea R Stan. Delivering on the promise of universal memory for spin-transfer torque ram (stt-ram). In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pages 121–126. IEEE Press, 2011.

[77] Mathias Noack. *Diploma Thesis Comparative Evaluation of Process Migration Algorithms*. PhD thesis, Dresden university of technology, 2003.

[78] Numonyx. Phase change memory.

[79] Venkata N Padmanabhan and Jeffrey C Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26:22–36, 1996.

[80] Themistoklis Palpanas and Alberto Mendelzon. *Web prefetching using partial match prediction*. University of Toronto, Department of Computer Science, 1998.

[81] Shien-Tai Pan, Kimming So, and Joseph T Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *ACM Sigplan Notices*, volume 27, pages 76–84. ACM, 1992.

[82] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Linearly compressed pages: a low-complexity, low-latency main memory compression

framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–184. ACM, 2013.

[83] Peter Pirolli, James Pitkow, and Ramana Rao. Silk from a sow's ear: Extracting usable structures from the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 118–125. ACM, 1996.

[84] James Pitkow and Peter Pirolli. Mining longest repeating subsequences to predict world wide websurfing. In *Proc. USENIX Symp. On Internet Technologies and Systems*, 1999.

[85] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology, 2009.

[86] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[87] Bogdan F Romanescu, Alvin R Lebeck, Daniel J Sorin, and Anne Bracy. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[88] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38, 2005.

[89] Ramesh R Sarukkai. Link prediction and path analysis using markov chains. *Computer Networks*, 33:377–386, 2000.

[90] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–197. ACM, 2013.

[91] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, Todd C Mowry, and Trishul Chilimbi.

Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 79–91. ACM, 2015.

[92] Junho Shim, Peter Scheuermann, and Radek Vingralek. Proxy cache algorithms: Design, implementation, and performance. *Knowledge and Data Engineering, IEEE Transactions on*, 11:549–562, 1999.

[93] Aidan Shribman and Benoit Hudzia. Pre-copy and post-copy vm live migration for memory intensive applications. In *Euro-Par 2012: Parallel Processing Workshops*, pages 539–547. Springer, 2012.

[94] James E Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[95] Clinton W Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 50–61. IEEE, 2011.

[96] P. Svard, J. Tordsson, B. Hudzia, and E. Elmroth. High performance live migration through dynamic page transfer reordering and compression. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 542–548, Nov 2011.

[97] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[98] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Prentice Hall Press, 2014.

[99] Patricia J Teller. Translation-lookaside buffer consistency. *Computer*, 23(6):26–36, 1990.

[100] Courtenay Vaughan, Mahesh Rajan, Richard Barrett, Doug Doerfler, and Kevin Pedretti. Investigating the Impact of the Cielo Cray XE6 Architecture on Scientific Application Codes. *2011 IEEE International Symposium on*

Parallel and Distributed Processing Workshops and Phd Forum, pages 1831–1837, 2011.

[101] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 340–349. IEEE, 2011.

[102] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[103] Carl A. Waldspurger. Memory resource management in VMware ESX server, 2002.

[104] KL Wang, JG Alzate, and P Khalili Amiri. Low-power non-volatile spintronic memory: Stt-ram and beyond. *Journal of Physics D: Applied Physics*, 46(7):074003, 2013.

[105] Paul R Wilson, Scott F Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the USENIX Annual Technical Conference*, 1999.

[106] Emmett Witchel, Josh Cates, and Krste Asanović. *Mondrian memory protection*, volume 30. ACM, 2002.

[107] HS Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[108] Wei Xu, Hongbin Sun, Xiaobin Wang, Yiran Chen, and Tong Zhang. Design of last-level on-chip cache using spin-torque transfer ram (stt ram). *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(3):483–493, 2011.

[109] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61. ACM, 1991.

[110] Tse-Yu Yeh and Yale N Patt. Alternative implementations of two-level adaptive branch prediction. *ACM SIGARCH Computer Architecture News*, 20:124–134, 1992.